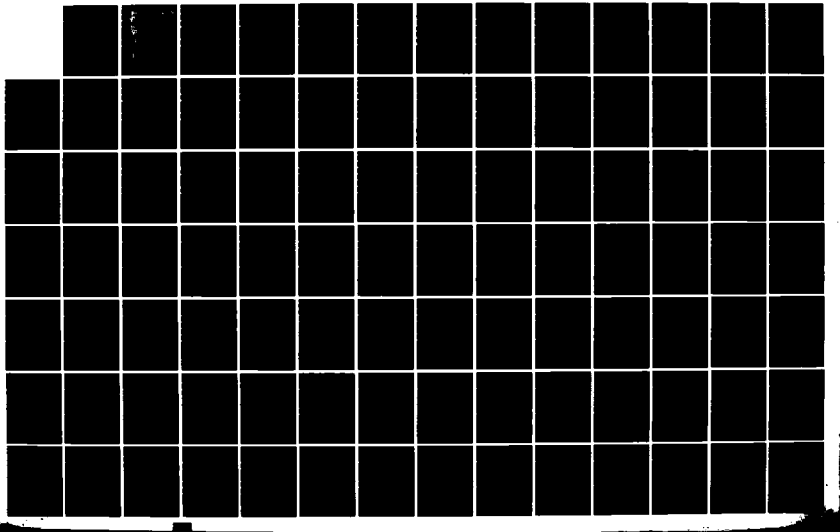
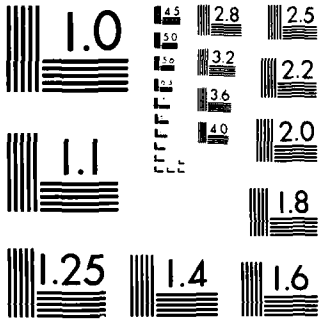
AD-A155 111 UNIFORM INTERFACES FOR DISTRIBUTED SYSTEMS(U) ROCHESTER 1/2
UNIV NY DEPT OF COMPUTER SCIENCE K A LANTZ MAY 80
TR-63 N00014-78-C-0164

UNCLASSIFIED F/G 9/2 NL





File Keith: Rochester

電腦科學

Uniform Interfaces for Distributed Systems

Keith A. Lantz
Computer Science Department
University of Rochester
Rochester, NY 14627

TR63
May 1980

Rochester

Department of Computer Science
University of Rochester
Rochester, New York 14627

Uniform Interfaces for Distributed Systems

Keith A. Lantz
Computer Science Department
University of Rochester
Rochester, NY 14627

TR63
May 1980

This report reproduces a thesis submitted to the University of Rochester in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

The preparation of this report was supported in part by the National Science Foundation under grant MCS-79-02971, by the Alfred P. Sloan Foundation under grant 78-4-15, and by the Defense Advanced Research Projects Agency, monitored by the ONR, under contract number N00014-78-C-0164.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: TR63	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Uniform Interfaces for Distributed Systems	5. TYPE OF REPORT & PERIOD COVERED: technical report
	6. PERFORMING ORG. REPORT NUMBER: 63
7. AUTHOR(s): Keith A. Lantz	8. CONTRACT OR GRANT NUMBER(s): N00014-78-C-0164
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Computer Science Department, University of Rochester, Rochester, NY 14627	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS: Defense Advanced Research Projects Agency, 1400 Wilson Boulevard, Arlington, Va. 22209	12. REPORT DATE: May 1980
	13. NUMBER OF PAGES: 155
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Office of Naval Research, Information Systems, Arlington, Va. 22217	15. SECURITY CLASS. (of this report): unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

command interaction	exception handling	network access
device-independence	message-passing	network operating systems
distributed computing	multi-processing	process management
distributed operating systems	multi-process structuring	protocols
distributed systems	networking	resource management

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

In designing a distributed system it is useful to distinguish between services provided to end-users and services provided to programs. The first type of service is provided by the user interface, the second by the system interface. Both user and system interfaces should gracefully extend the local (single-machine) computing environment to embrace remote resources; they should provide a coherent view of the distributed system. This report presents several mechanisms and paradigms for building distributed systems with uniform interfaces.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. tools
user interfaces
virtual terminals
interfaces
interprocess communication

20. These techniques are discussed in the context of a multiple-machine, multiple-network distributed system, called RIG, developed at the University of Rochester over the last six years. Logically, RIG can be thought of as a collection of independent processes running on various computers and cooperating via messages. Typical operating system functions, such as file access, terminal communication, and printing, are provided by server processes associated with each system resource. Each server is responsible for maintaining its own resource-specific communication protocol with its resource and for providing a standard message interface to other RIG processes. Thus, the distinction made in traditional systems between operating system services and user processes has been abandoned in favor of a uniform message interface.

Building on this foundation, this report presents four contributions to research in distributed systems:

1. Virtual Terminals are presented as the means for managing a large number of application programs per user.

2. Principles of command interaction that facilitate the use of distributed resources are outline.

3. Mechanisms for process management in a distributed system are presented.

4. Paradigms are presented for how processes should be written and communicate.

Acknowledgments

RIG could not have become a reality without the efforts of all members (past and present) of the RIG Working Group, especially Eugene Ball, Edward Burke, Ilya Gertner, Klaus Gradischnig, and Richard Rashid. The patience, good humor, and technical expertise of Richard Rashid, put to use through countless hours of close collaboration writing both programs and papers, are particularly appreciated.

I would like to thank my advisors — especially Christopher Brown and Jerome Feldman -- for their patience in reading and correcting several drafts of this thesis, and for contributing greatly to its overall structure. Elaine Mackoff and Rose Peet may be credited for any consistency of style that emerges in the manuscript.

Abstract

In designing a distributed system it is useful to distinguish two types of system service: those services provided to end-users and those provided to programs. The first type of service is provided by the user interface, or user-system interface. The second type is provided by the system interface, or program execution environment. Both user and system interfaces should gracefully extend the local (single-machine) computing environment to embrace remote resources; they should provide a coherent view of the distributed system. This thesis presents several mechanisms and paradigms for building distributed systems with uniform interfaces.

These techniques are discussed in the context of a multiple-machine, multiple-network distributed system, called RIG, developed at the University of Rochester over the last six years. Logically, RIG can be thought of as a collection of independent processes running on various computers and cooperating via messages. Typical operating system functions, such as file access, terminal communication, and printing, are provided by server processes associated with each system resource. Each server is responsible for maintaining its own resource-specific communication protocol with its resource and for providing a standard message interface to other RIG processes. Thus, the distinction made in traditional systems between operating system services and user processes has been abandoned in favor of a uniform message interface.

Building on this foundation, this thesis presents four contributions to research in distributed systems:

1. Virtual Terminals are presented as the means for managing a large number of application programs per user. Any number of Virtual Terminals may be mapped to a physical device simultaneously, and each Virtual Terminal may be written to or queried for user input. In addition, the Virtual Terminal Management System provides extensive facilities for editing text, the ability to save all output on stable storage, and sophisticated mechanisms for the management of screen space. Virtual Terminals allow application programs to remain unaware of the specific physical device through which they are communicating.

2. Principles of command interaction that facilitate the use of distributed resources are outlined. Tools are logically (and physically) separated into user interface and service processes. Table-driven command interpreters enforce a consistent command interaction discipline, isolating the user from the idiosyncrasies of each tool. Together with the Virtual Terminal Management System, the command interface presents an elegant, robust, and consistent interface between RIG and the user.

3. Because most traditional operating system services are associated with server processes, resource management is viewed fundamentally as a problem of process management. Processes may be created "by name," and registration facilities enable a process to register its interest in, for example, the death of any other process (see contribution 4). An explicit process tree is used to group processes created in response to particular user requests or jobs. The process tree, together with the registration facilities, simplifies the deallocation of resources associated with terminated jobs.

4. Paradigms are presented for how processes should be written and communicate. Distinctions are drawn between dedicated and multiplexed servers, and between three modes of interprocess communication -- atomic transactions, connections, and asynchronous emergency messages. Emergency messages, in particular, provide a simple yet powerful mechanism for handling inter-process exceptions: Registration facilities and event handlers enable a process to register its interest in exceptional events that occur with regard to any other process; notification of the occurrence of an event is by emergency message.

Table of Contents

List of Figures

Foreword

RIG has been developed over the past six years by a team of as many as ten designers and implementers. The original design was proposed in 1974-75 by Eugene Ball, Jerome Feldman, James Low, Paul Rovner, and Richard Rashid. I became involved with RIG in the summer of 1975, and, since then, have been involved to varying degrees in every aspect of its development.

It is sometimes difficult, therefore, to separate my work distinctly from that of others in the RIG Working Group. Parts of this thesis were, in fact, written with the assistance of other members of that group. Much of Sections 2.2, 6.2, and 6.3 was written with Eugene Ball, Edward Burke, Ilya Gertner, and Richard Rashid and presented at the 1979 Computer Networking Symposium [12]. Much of Sections 2.3.1 and Chapter 3 was written with Richard Rashid and presented at the 7th Symposium on Operating Systems Principles [129]. Pieces of the remaining text were adapted from a variety of internal memos written with Eugene Ball and Richard Rashid. In all instances I was both a principal author and an architect of the ideas presented.

CHAPTER 1

Introduction

A single computer system cannot be all things to all users. Some systems are better suited for numerical applications, some for symbolic manipulation, some for parallel computation. Some systems provide a better user environment. (Here, a computer system is composed of the machine and all attendant software.)

If a user wishes to use different computer systems for different tasks at different times, it is desirable to provide him with a coherent interface to his available pool of resources, that is, to isolate him wherever possible from the idiosyncrasies of individual computer systems. Alternatively, given an application program that requires a mix of features or that can perform certain actions in parallel, it may be most effective to employ more than one computer system. These two considerations have served as the primary motivations behind the development of computer networks and distributed systems.

At the University of Rochester we have had six years of experience in the design and implementation of a multiple-machine, multiple-network distributed system called RIG. RIG was built to serve as an intermediary between the human user (working through a display terminal or personal computer) and a variety of computer systems. The bulk of the user's computational requirement is met by these systems, which are either partially integrated into the RIG system through a fast local network or loosely coupled to it through the ARPANET. RIG also provides a number of basic services such as printing, plotting, local file storage, and text-editing.

In designing RIG, it was useful to distinguish two types of system service: those services provided to end-users and those provided to programs. The first is provided directly to the user through a user interface, or user-system interface. The user obtains these services by typing commands or requests that are satisfied by actions initiated by the user interface. The second type of service is provided by programs or processes executing on behalf of the user. The user's program obtains these services by executing "system calls." The system software that interprets and satisfies these calls implements the system interface, or program execution environment [214].

a base-level NOS is distributed operating system (DOS) [109].

3. meta-resources

Functions common to a broad range of applications should be made available in a fashion that eliminates their being designed and implemented again and again. What are the common attributes that can be abstracted from different implementations of similar objects in order to permit interchangeable use of resources? Should the NOS provide access to both meta- and local resources separately?

4. visibility of distribution

Inherently, it appears to be a good idea to remove the burden of distribution from the user, but in currently available systems, performance frequently decreases when access to remote resources is required. Furthermore, certain NOS operations may be supported only for entities that reside on the same host. The most appropriate design may be to provide users with mechanisms to influence, if not specify, resource patterns.

5. reliability

Reliability measures are expensive, but replication of critical system data bases can allow continued operation in the presence of localized failures. Application programs should be provided with mechanisms that can be used to build reliable services for an environment where the system configuration can change dynamically due to failures.

6. resource allocation, management, and access control

What entities are authenticated, who performs the authentication, and how often? Who makes the resource selection: the user, the system, or a combination? Is the decision made on the basis of dynamic network conditions, or on the basis of static user profiles? At what point does the NOS relinquish control to the local host mechanisms? If resource management is logically and physically centralized, the implementation is simplified since the effect of the distributed environment is minimized, but the system is vulnerable to failures of the site supporting the control function and performance may be degraded due to the necessity for interhost communication each time the function is needed. By replicating the same control function at many sites both of these problems are reduced. For fully distributed control each entity needing a control function has its own implementation of the function, thus breaking the dependence of one host on another and providing better performance, but adding to the expense of design, implementation, and operation.

equipment and the system-dependent protocols that complicate interaction with distributed resources.

Network access machines are intended to improve the user interface by automating the access functions. This is typically done by associating executable command files with each accessible resource; when an attempt is made to access the resource, the NAM reads the file to generate the appropriate system-dependent commands to the remote host. A NAM is meant to provide user access to remote hosts without necessitating any changes in the hosts' hardware or software. It can interact with the constituent hosts at the user interface only.

2.1.2 Network Operating Systems

A network operating system, on the other hand, extends remote access to the programming environment. Both users and programs are allowed to access resources without regard to their physical location. The intent is to provide a view of the network similar to that provided by a traditional operating system for a single computer, that is, the entire ensemble of machines appears as a single entity.

A NOS requires that additional hardware or software be provided for each host. Important issues to be addressed when designing a NOS include (after [214, 228]):

1. mission-oriented vs. general-purpose system

   Is the system meant to solve a given, well-defined problem formulated in terms of specific control and data requirements with real-time or reliability constraints? Or is it meant to be an "open ended" information processing utility that supports a wide variety of application domains?

2. base level vs. guest level implementation

   Can the system be built from bare hardware components such that the NOS can be specifically designed to function effectively and efficiently together? Or is it necessary to utilize existing operating systems, or minor variants thereof, such that the NOS acts principally to coordinate the activity of the operating systems to provide an integrated computational facility? If the guest-level approach is taken and application processes are allowed to run programs without regard for the expanded accessibility to NOS resources, an encapsulation interface must be provided that can examine host system calls to redirect calls to NOS modules. Alternatively, programs may be programmed to run in the NOS environment and to use the set of primitive operations supplied by the NOS for accessing network resources. If a base-level approach is taken, access to local services should be as efficient as on existing non-distributed operating systems. A more reasonable term for

The gateway machines also provide a number of basic services such as printing, plotting, local file storage, and text-editing. Devices supported include a 256x512 color raster graphics display, a tape drive, an electrostatic printer-plotter, a drum scanner, and various display terminals. Together, the Eclipses provide 600 MBytes of primary disk storage.

In general, the user accesses these facilities through a display terminal connected to the gateway machines or through a personal computer. Certain functions (such as file transfer) are provided directly to users on the two time-sharing systems. RIG thus provides many of the functions attributed to gateways, network access machines, and network operating systems. To better understand the functions RIG(ITS) is intended to provide, I will first present a taxonomy of distributed systems, followed by an overview of RIGITS.

2.1 A Taxonomy of Distributed Systems

Distributed systems may consist of tightly coupled (shared memory, centralized resource management) or loosely coupled (no shared memory, completely autonomous) functional units. Examples of the former are multi-processors such as the CRAY-1 and Illiac IV; examples of the latter are heterogeneous networks such as the ARPANET. In between are multi-processors such as Hydra and Cm*, homogeneous networks such as MININET, and systems such as RIG.

RIGITS is concerned with computer networks, which come in several forms. Remote-communication networks simply move information from one place to another. Resource-sharing networks allow resources on one computer system to be shared by other systems in order to reduce costs and provide remote access. Distributed-processing networks allow several autonomous computer systems to solve problems by division of labor or functional specialization. As such, distributed-processing networks are a natural extension of real-time multi-programming systems. Distributed-processing networks are built (logically) on top of resource-sharing networks, just as resource-sharing networks are built on top of remote-communications networks (typically referred to as communications subnetworks).

Networks may be homogeneous (all hosts of the same type) or heterogeneous (hosts of different types). They may be geographically close (all machines near each other) or distant (geographically distributed).

2.1.1 Network Access Machines

Resource-sharing and distributed-processing networks provide the basis for network access machines (NAM) and network operating systems (NOS), respectively. Both NAM's and NOS's are intended to isolate the user as much as possible from the idiosyncrasies of the physical

CHAPTER 2

An Architecture for Distributed Systems

Distributed systems may be built on a number of bases: traditional multi-processing operating systems, multi-processors, networks, and networks interconnected by gateways. RIG, in particular, connects three networks via a dual-processor gateway (see Figure 1).

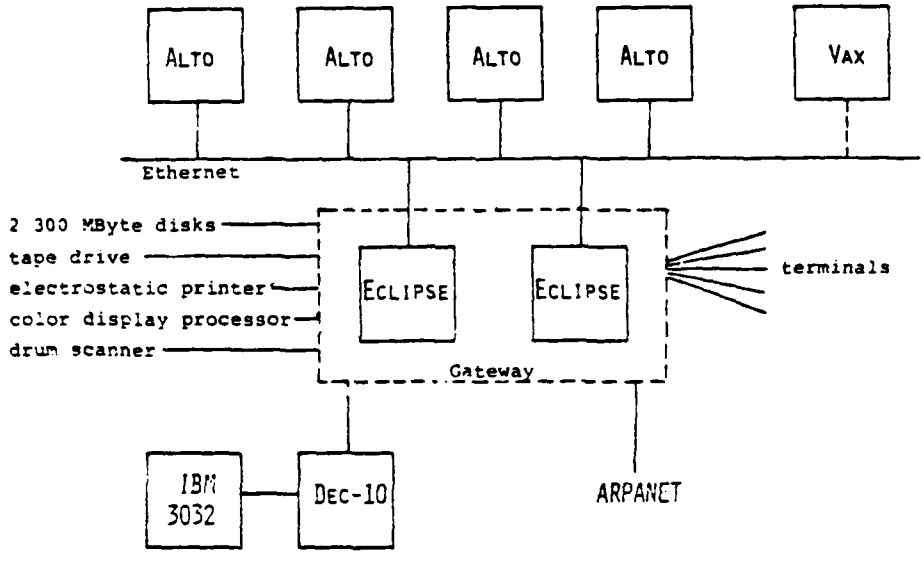


Figure 1. The RIG environment.

The bulk of the user's computational requirements is met by two time-sharing systems (DEC-10/TOPS-10 and VAX/UNIX) and four personal computers (Xerox Altos -- 16-bit 64 KW minicomputers with 606x808 frame-buffer raster-scan displays [212]). The Altos, gateway machines (Data General Eclipses), and the VAX are connected via a 3 MHz broadcast network (Ethernet [149]). The DEC-10 (campus network) communicates with the gateway over 9600 baud asynchronous and 50 KHz synchronous lines. The gateway is connected to the ARPANET as a Very Distant Host via a 50 KHz synchronous line.

nal Management System was inspired primarily by Swinehart's work on debugging [207]. NLS/Augment and related work was the primary contributor to the RIGITS command interface [6, 63, 64, 105, 226]. Related work at the Augmentation Research Center on the Command Meta Language and a proposed Frontend for the National Software Works is also important [7, 8, 9, 106]. RIG(ITS) is further described in [12, 13, 125, 128, 129, 184, 185]. Related research at Rochester in distributed systems is described in [14, 71, 72, 73, 82, 83, 161].

Notable single-processor multi-process operating systems include the RC4000 [29, 30] (perhaps the first operating system to employ message-passing), the GEC4000 series [80], Cal [124, 205], SUE [99], TENEX [24] (and its son, TOPS-20), UNIX [21], Elf [179], MERT [140], DEMOS [17], Thoth [39, 40], NLTSS [61], and Pilot [175].

Much of the important work on distributed systems has involved Bolt Beranek and Newman Inc., including the ARPANET [186], RSEXEC [46, 213], and the National Software Works [66, 81, 96, 151, 162, 195, 196].

Although the RC4000 may lay claim to being the first message-passing operating system, DCS wins the sweepstakes for one of the first, best, and most long-lived message-based network operating systems [67, 68, 69, 70]. Watson and Fletcher present a design for a network operating system (hereafter called LLLNOS, for Lawrence Livermore Laboratory Network Operating System) which comes as close as any to RIGITS [77, 227, 228, 229, 230]. Other systems include DCN [150], DPS [231, 232], Hydra [42, 134, 235], DCCS [60], MININET [138, 145], XNOS [115, 116, 117, 118], the Honeywell Distributed Processor [107], Roscoe [199, 200], MARS [119], StarOS [112], and Medusa [163].

Network access systems include NAM [76, 188], ANTS [27], REX [22], Elf [178], RITA [4, 5], and various adaptations of UNIX [41, 144].

Virtual Terminal-like facilities are provided by such varied systems as ITS [62], ZONES [139], TTDL [120], the TSO Job Session Manager [147], and Virtual Terminal UNIX [159]. Model [154] has recently extended Swinehart's work by employing DLISP [201, 210]. Representative network virtual terminal protocols are discussed in [15, 18, 19, 49, 53, 54, 65, 104, 142, 157, 192, 193].

USC/Information Sciences Institute has done important work in user interfaces, initially under the title of the Information Automation Project and now known as SIGMA [1, 88, 89, 189, 203, 204]. Sophisticated command languages have also been provided by UNIX, TENEX, TOP-20, VAX/VMS, and numerous other operating systems.

Two of the best references on distributed systems architecture and distributed operating systems are [121, 214]. Other relevant tutorials include [2, 23, 51, 52, 146, 148, 215].

Each of the four chapters constituting the body of the thesis addresses one of the four major contributions presented in Section 1.4, and are relatively independent. Chapter 3 discusses virtual terminal management. The concepts presented are applicable to any computer system, whether or not it is distributed or based on processes communicating via messages. The particular implementation for RIG is presented.

Chapter 4 discusses the use of tools (application programs) and the command interface. The principles of command interaction, command profiles, and user profiles are independent of the implementation and computing environment. The abstract notions of tools and their associated Tool Interface Processes are, however, inspired by a distributed environment where tools may run on distant hosts. Much of the command interface has not yet been implemented, but the basic principles are evident in the work that has been completed (Chapter 7).

Chapter 5 discusses resource management. Since all RIGITS resources are managed by processes, the fundamental problem is one of process management. The techniques for process management are applicable in any process-based environment. Authentication and network job control are briefly discussed.

Chapter 6 presents some conventions for process structuring and intercommunication that have proven invaluable in RIG. These include the various protocols employed for handling connections and exceptional conditions. These protocols correspond to Levels 5 and 6 of the ISO Reference Standard for Open Systems Architecture [58, 236], and are recommended for any RIGITS-like system.

Chapter 7 comes back to earth to discuss RIG, an example of a system built on the RIGITS abstraction. The differences between RIG and RIGITS are discussed and an extended example is presented, involving concepts from the preceding chapters.

Chapter 8 concludes the thesis with a discussion of successes, failures, and future work.

1.6 Related Work

Many of the ideas embodied in RIGITS derive from, or parallel, experience gained by the Computer Science community with user interfaces, command languages, operating systems, distributed computing, and network protocols. Historical perspectives in each of Chapters 2 through 6 will indicate the influential and related work in particular areas. Here I will briefly list the appropriate references for each system, in order to eliminate the clutter in subsequent sections. A more comprehensive version of the bibliography contained in this thesis may be found in [126].

The RIGITS message-passing paradigm grew out of experience with interprocess communication facilities developed for the Stanford Hand-Eye Project [75] and the work of Walden [221]. The Virtual Termi-

4. Paradigms are presented for how processes should be written and communicate. Distinctions are drawn between dedicated and multiplexed servers, and between three modes of interprocess communication -- atomic transactions, connections, and asynchronous emergency messages. Emergency messages, in particular, provide a simple yet powerful mechanism for handling inter-process exceptions: Registration facilities and event handlers enable any process to register its interest in exceptional events that occur with regard to any other process; notification of the occurrence of an event is by emergency message.

The above contributions are not as disparate as they may appear. I was first concerned with the user interface — command interaction and Virtual Terminals. Given the distributed computing environment, that interface had to be implemented with processes. Moreover, any tools with which the user might interact were also implemented as processes. Because a large collection of processes and other resources are associated with each user, it was necessary to provide facilities for resource management and to develop a practical methodology for distributed computing (multi-process structuring).

In addition, RIGITS demonstrates the importance of declarative profiles as a means of tailoring the system at run-time. Many system entities, such as process types, terminals, hosts, tools, and users, are defined via profiles. A process that deals with such an entity is not written in terms of a specific instance of the entity (such as a particular terminal), but rather in terms of a generic prototype whose characteristics may be "bound" at run-time by a profile (such as a Virtual Terminal). While this technique seems an obvious extension of table-driven parsing, parallels to artificial intelligence techniques are also apparent: Profiles are declarative knowledge; processes represent procedural knowledge. Processes are written in terms of generic objects, and manipulate (or are driven by) instantiations thereof. It is fitting that an "intelligent" software system should benefit from (and contribute to!) work in the representation and use of knowledge.

1.5 A Roadmap

Chapter 2 presents an overview of RIGITS and, in the process, sets forth most of the terminology used in the remainder of the thesis. The system architecture and user interface are discussed. The notion of declarative profiles for run-time tuning of the system is introduced. Should the reader become overwhelmed by the terminology, he might do well to glance through Chapters 3 through 7 (particularly Chapter 7) as he reads Chapter 2.

Modular decomposition into processes communicating solely via messages centralizes the access to each resource and leads to a high degree of security. Messages provide a wide range of synchronization strategies and a uniform mechanism for conveying control or data information and signaling exceptions. Semaphores, by comparison, can distribute both access and synchronization throughout the code executed by many processes. Monitors centralize the access to a resource but distribute the use of these functions over all processes. Neither monitors nor semaphores allow the resource to do its own scheduling. Lastly, even if a monitor model is used in a distributed system, the reality is that messages are being exchanged (between machines). Therefore, distributed systems research at Rochester adhers to a strict message-passing view. (The pros and cons of message-passing versus monitors are discussed at length in, for example, [130].)

1.4 A Summary of Contributions

Many people have been involved in the design and implementation of RIGITS (see the Foreword). This thesis describes four contributions for which I was largely responsible:

1. Virtual Terminals are presented as the means for managing a large number of application programs per user. Any number of Virtual Terminals may be mapped to a physical device simultaneously, and each Virtual Terminal may be written to or queried for user input. In addition, the Virtual Terminal Management System provides extensive facilities for editing text, the ability to save all output on stable storage, and sophisticated mechanisms for the management of screen space. Virtual Terminals allow application programs to remain unaware of the specific physical device through which they are communicating.

2. Principles of command interaction that facilitate the use of distributed resources are outlined. Tools are logically (and physically) separated into user interface and service processes. Table-driven command interpreters enforce a consistent interaction discipline, isolating the user from the idiosyncrasies of each tool.

3. Because most traditional operating system services are associated with server processes, resource management is viewed fundamentally as a problem of process management. Processes may be created "by name," and registration facilities enable any process to register its interest in, for example, the death of any other process (see contribution 4). An explicit process tree is used to group processes created in response to particular user requests or jobs. The process tree, together with the registration facilities, simplifies the deallocation of resources associated with terminated jobs.

user cited above, as well as a language-oriented "programming environment." To the programmer as implementer, distributed computing becomes attractive only if it gracefully extends the local program execution environment to embrace resources on other machines.

In particular, a sophisticated distributed system should:

1. encourage the use of separate modules for distinct modules with well-defined functions (modular decomposition);
2. allow for incremental addition of new processors, devices, and services;
3. place no a priori restrictions on which modules can communicate with each others;
4. provide for location independence;
5. provide uniform access to basic resources such as file systems, printers, and terminals;
6. support resource simulation and encapsulation;
7. provide convenient mechanisms for handling errors and other exceptions, both within a single module and between modules;
8. support both centralized and decentralized services.

The goal is to create an environment in which modules may communicate without concern for the topology of the system as a whole, using as few mechanisms as possible.

Logically, RIGITS can be thought of as a collection of independent processes running on various computers and cooperating via messages. Each process has a distinct logical address and performs a specific set of functions, as defined by its interface. Typical operating system services, such as file access, terminal communication, and printing, are provided by server processes associated with each system resource (such as files, terminals, and data bases) [111]. A server defines the abstract representation of its resource and the operations on this representation. A resource may only be accessed or manipulated through its server(s). Because servers are constructed with well-defined interfaces, the implementation details of a resource are of concern only to its server(s).

All communication between processes takes the form of messages. A message is the smallest unit of data that must be exchanged for a meaningful action to take place. No local variables can be directly examined and no procedures directly invoked by another process without an explicit message request. Because shared memory is not used, there is no distinction between local (intra-host) and remote (inter-host) communication.

being debugged and an editor without destroying the state of either program. For systems programmers, the ability to display simultaneously the state of programs running on different machines is indispensable in debugging network software.

In short, a sophisticated distributed system should provide:

1. a consistent command interaction discipline across all available application programs;

2. support for the creation and handling of a large number of application programs per user, and facilities for managing their input and output;

3. a terminal input/output interface to application programs which is independent of particular physical devices;

4. fast response to user interaction;

5. facilities for tailoring the user interface to each user's preferences and needs.

Traditional systems fail to satisfy these goals. Typically, the user has only one logical line of communication. Input and output are both multiplexed in time, forcing the user periodically to look in on each program to assess its state. His working context is at best a long scroll of paper, and at worst his fragile short-term memory. Moreover, each tool (subsystem) employs its own special-purpose facilities to interact with the user in idiosyncratic ways.

RIGITS, on the other hand, gives its users the freedom to perform any number of activities simultaneously. A user sitting at his display terminal may view the output of various application programs on different areas of his screen. He may rearrange his display, edit or save its contents, or direct keyboard input to any of the programs under his control. Table-driven command interpreters serve to isolate the user from the idiosyncrasies of each tool. User profiles allow him to tailor the interface to his own needs. To ensure fast response and support the encapsulation of existing services, tools are separated into user interface and service components. These facilities combine to present an elegant, robust, and consistent interface between RIGITS and the user.

1.3 The System

Just as computer systems should respond to the needs and desires of end-users, they must also provide an environment that encourages and facilitates the work of the programmer. To the programmer as user, this means that he should be provided with all the amenities of the turn-key

> It is advantageous to remain faithful to the current design and implementation of [RIG] in our discussion so that remarks are supported by implementation, testing, and experience. It is also advantageous to include how we now believe the system should have been done, drawing on the benefit of hindsight and experience. It is equally advantageous to abstract the discussion with a particular system to provide wider applicability of our conclusions. All three of these competing goals govern this report; we trust the reader will recognize the different tacks in the course of the discussion. [39, p. 3]

1.2 The User

Computer systems, distributed or not, should respond to the needs and desires of their users. They should provide a working environment tailored to the methods and habits of the individual, one in which the computer serves to expand rather than restrict the freedom to think, create, and act.

The user's primary concern is to get a job done, as simply as possible. The user has little or no interest in the peculiarities of the different systems to be used, such as the syntax and semantics of the various command languages. Naming, protection, accounting, and access procedures should be as uniform as possible. Standard error diagnostic and recovery services should be provided to isolate the user from system-dependent error messages. On-line assistance should be easily accessible. At best, the command language provided should allow maximum access to the features of all the systems available with minimum system-dependent interaction [76].

Moreover, the user should be allowed to perform multiple tasks simultaneously:

> Being able to switch back and forth between tasks results in a relaxed and easy style of operating more similar to the way people tend to work in the absence of restrictions. To use a programming metaphor, people operate somewhat like a collection of coroutines corresponding to tasks in various states of completion. These coroutines are continually being activated by internally and externally generated interrupts, and then suspended when higher priority interrupts arrive, e.g., a phone call that interrupts a meeting, a quick question by a colleague that interrupts a phone call, etc. ...it is of great value to the user to be able to switch back and forth quickly between related tasks. [210, p. 3]

It is invaluable, for example, to be able to switch between a program

These deficiencies derive from the fact that traditional network architectures do not support (the evolution of) a network or distributed operating system. The ideal distributed operating system would provide uniform and controlled access to distributed resources; it would act to mediate incompatibilities among the resources so that they can be used together; and it would provide an environment for uniform accounting and administration. In short, a distributed operating system should provide the same sort of interface to the network as a traditional operating system provides to its computer.

The key to a successful distributed operating system is a modular and layered design [164, 228]. Associated with each layer N are two interfaces defining the set of services provided to layer N+1 and the set of services required of layer N-1. Layers only interact through their interfaces. The services of layer N may be further decomposed into modules (processes). Modules, like layers, provide at their interface a well-defined set of services, and their internal implementation is not visible on the other side of the interface.

An interface, then, can be defined as a set of conventions for the exchange of information between two entities [228]. It consists of three components:

1. A set of visible abstract objects (such as files or virtual terminals) and for each a set of allowed operations and associated parameters.

2. A set of rules governing the legal sequences of these operations.

3. The encoding and formatting conventions required for operations and parameters.

In the literature, distinctions are often drawn between the terms protocol and interface. In general, this thesis follows Watson [228] in regarding these terms as synonymous.

The use of layers and modules communicating only through well-defined interfaces allows complex systems to be broken down into more easily understood pieces. The correct operation of these pieces may in turn be more easily verified. Alternate services provided by one layer (module) may share the services provided by other layers (modules). The system can evolve more easily because the algorithms and mechanisms implementing a given layer (module) can be changed without affecting the service offered, providing the service offered at the interface remains unchanged.

The body of this thesis is based on a distributed system architecture known as RIGITS, for RIG In The Sky. I will return to the current implementation, RIG, in detail in Chapter 7. Any attempt to discuss both an abstraction and an implementation can lead to difficulty; in discussing Thoth, Cheriton phrased it well:

6. the ability to handle increased complexity — achieved by decomposition of the problem-solving task into subtasks, each reduced in the range of possible activity as compared with the overall task.

However, at present the amount of resource sharing and distributed computing occurring on computer networks falls far short of that which is possible. Traditional network architectures consist of a set of function-oriented protocols, such as virtual terminal and file transfer protocols, built on top of an interprocess communication protocol [35, 44, 47, 48, 170, 171]. The full potential of computer networking cannot be realized with such an architecture for the following reasons (after [214, 228]):

1. The mechanics of access are difficult.

   To make effective use of the network resources a user must master network access mechanisms as well as the operating system for each host providing a resource he wishes to use. He must log in to his local host, use a network access program, and then log in to his target host(s), each possibly using different conventions. There is generally no single source that can be consulted for information about available resources. Consequently, users are often unaware of the resources available to them. Even after learning that a particular resource exists and even if the resource is well documented, a user must often rely on word-of-mouth folklore from other users to learn how to use it.

2. The resources provided by the various hosts are generally incompatible with each other.

   A user often encounters great difficulty in attempting to use individual resources for the various hosts together in an integrated fashion. To use the output of a program on one host as the input to a program on another host, the user must manually invoke the appropriate data transfer and transformation functions. No basis is provided for easily creating, in a layered fashion, new resources or services out of existing ones. Each programmer desiring to provide or use a new network-sharable resource must face anew all the issues of data type translation, command and reply formatting and parsing, naming, protection, and interfacing to the transport protocol layer.

3. Accounting and other administrative procedures are awkward.

   A user must deal with each administration controlling a host that manages a resource he plans to use.

In RIG equal emphasis was given to user and system interfaces. Each was designed to gracefully extend the local (single-machine) environment to embrace distributed resources, that is, to provide a coherent view of the distributed system. This thesis presents several of the mechanisms and paradigms developed for building distributed systems with uniform interfaces.

1.1 Motivation

The area of distributed systems is new, and, as in any new field, there is a lack of agreed upon terminology. Because distributed systems have been built on so many bases -- multi-processing operating systems, multi-processors, networks, gateways -- I will loosely define a distributed system to be any computer system (or collection of computer systems) that allows a program to be written as a collection of cooperating processes. A process is a self-contained collection of code and data segments; it may be understood informally as a procedure running on a real or virtual processor. These processes may be spread out among a variety of processors, which may in turn reside on a variety of networks. (Using this definition, a multi-processing operating system is simply a degenerate case of a distributed system.) RIG, in particular, is composed of three networks connected by a single gateway (see Chapter 2).

At their best, distributed systems can provide (after [131]):

1. lower communication costs — achieved by abstracting (preprocessing) data for transmission (lowering communicating bandwidth requirements) and by placing processing elements near the data (reducing the distance data must be transmitted);

2. lower processing costs — achieved through the use of cheaper, less complex processing elements which can be mass produced and through load-sharing (allowing relatively idle processing elements to handle some of the work of a busy processing element);

3. an increased repertoire of resources -- achieved with the addition of new host types;

4. enhanced performance -- achieved through parallelism, load balancing and functional specialization, and through the placement of processors near sensing devices and devices to be controlled;

5. increased reliability and flexibility — achieved through redundancy in communication paths and processing elements, and through modularity of design;

7. extensibility

   Is it possible to write user programs without intervention by systems programmers, or must the user make do with the services provided? If users can build new services, an NOS can start with a few services and evolve from there. New resources can be constructed out of existing ones without introducing new privileged (systems) programming. Systems desiring to participate in the NOS can do so with minimal implementations.

8. interference between NOS and non-NOS activity

   If the NOS is implemented as a base-level system, all activities are NOS activities. However, a guest-level system should not conflict with non-NOS activity on autonomous hosts.

9. administration

   Either a centralized authority or increased user sophistication is required.

2.1.3 Gateways

The interconnection of networks is a topic of increasing importance (see, for example, [36, 206]). Networks are connected via gateways that provide the necessary protocol transformations. In RIG, for instance, the gateway Eclipses perform protocol transformations to enable Altos on the Ethernet to communicate with the DEC-10.

Building an operating system that spans gateways is facilitated by current work on internetwork protocols [25, 168, 169]. Violet, for example, is a decentralized application program spanning an internetwork [123].

2.1.4 The Place of RIGITS

Following the taxonomy outlined above, RIGITS possesses the following characteristics:

1. general-purpose system composed of loosely coupled, heterogeneous components

2. base level implementation on the gateway Eclipses, with guest-level extensions to the Ethernet environment and the DEC-10

3. meta-resources across the Ethernet

4. visibly distributed file system; transparent access to gateway services

5. reliability based solely on mutual suspicion -- no replication

6. decentralized resource management with user and system resource selection

7. extendable program set

8. only DOS activity on the gateway machines; NOS activity on the DEC-10 impacts upon the performance of TOPS-10

9. decentralized administration

RIGITS is first and foremost a system meant to provide access to all available resources through a single terminal. Where it cannot encapsulate particular hosts or networks RIGITS attempts to provide more limited network access services. Where these mechanisms cannot be provided, the user and programmer must fall back on system-dependent interaction (although the Virtual Terminal Management System is always available). Most of this thesis discusses RIGITS from the viewpoint of the gateway machines, that is, as a distributed operating system.

2.2 The RIGITS System Architecture

2.2.1 System Superstructure

Logically, RIGITS can be thought of as a collection of independent processes running on various computers and cooperating via messages. The software on a given RIGITS machine, or host, is organized in two logical layers: a Kernel and a set of processes. The Kernel provides the support functions of message-passing, process scheduling, physical memory management, and interrupt handling. At the interface to the Kernel there is no concept of a connection or link, only the ability to send and receive messages. The basic process synchronization mechanism is provided by the Kernel, where a process can wait or not, at its option, for a particular send or receive to complete.

Processes provide both system and user (application) services. Although some processes are servers, they are no different than any other (application) process in terms of protection or access to system calls. Indeed, a given process can operate in either or both server and user roles at different times: A server, PA, may require the service of another server, PB, in which case PA is operating effectively as a user process. Resource naming, connections, and message semantics are provided at the process level.

Each host supports its own complement of server and user processes. These processes typically include:

1. a Process Manager
2. a Job Manager
3. servers for local file systems, networks, printers, terminals, spoolers, and so on
4. a collection of Virtual Terminal Controllers, Monitors, Executives, tools, and Command Interpreters on a per user basis

Servers need not be permanently associated with a particular host. A host typically provides a fixed set of services, determined when the system is initialized, but can be dynamically reconfigured under certain situations. RIGITS does not support process migration; when a server is moved to another host, any state (old connections) associated with the old server will not be carried over to the new instantiation. However, services are requested by name (see Section 2.2.3.3), so new connections will be opened with the appropriate process. The services provided by a given host are indicated in its Host Profile (see Section 2.4).

2.2.2 Resource Management

Because all resources are managed by processes, the fundamental problem of resource management is process management. On each RIGITS host, a Process Manager provides for the creation, destruction, and registration of processes. Processes may be created "by name" -- given a name like "Executive," the Process Manager will create an instance of the Executive based on a Process Profile. Registration facilities allow a process to learn of exceptional events that occur to any other process.

The Job Manager is responsible for managing the terminals and users "connected" to the host. It also provides authentication and access control services where necessary.

Resource management is the subject of Chapter 5.

2.2.3 Interprocess Communication

RIGITS processes communicate with each other solely by sending messages. A message consists basically of:

- source address
- destination address
- message id
- typed data

Message ids are defined system-wide and indicate the function to be performed by the receiver. Messages are variable-length and may be uniformly represented by name-value slots as in PLITS [71].

Messages are queued separately by the Kernel for each destination and passed to the receiving process as they are requested. A destination in RIGITS is specified by a process-port pair, where a port is simply a sub-address within a process. Although the Kernel places no restrictions on access to ports, most server processes employ specific ports to communicate with specific customers. A process is free to assign different priorities to each port, to select a specific port on which to receive messages, to change the message queuing capacity of a port, or to lock a port such that messages destined for that port will be queued but not returned in response to a receive. In short, a process uses ports for selective message reception, flow control, and multiplexing.

An important variant of ports has been used in several systems, including a descendant of RIGITS for UNIX [174]. Ports are disassociated from processes, providing for process migration. Processes access ports via capabilities, providing increased protection [110].

Three aspects of the communication techniques used in RIGITS eliminate the need to know the actual location of services in the distributed system:

1. all basic services are provided by RIGITS processes through the use of messages (no shared memory);

2. message transmission is transparent between machines;

3. inter-process communication can be initiated by name.

The following three sections discuss each point in turn.

2.2.3.1 Access to Basic Services -

Standardized message protocols provide independence from the location and idiosyncrasies of each resource. They allow processes to treat message-passing as remote procedure calls or pipes, and provide for asynchronous event handling. These protocols include consistent

mechanisms for opening, closing, reading, and writing entities such as files, virtual terminals, and line printers.

Since message primitives are indivisible from the standpoint of the process, they behave like procedure calls. Together with reliable transmission and flow control this approach relegates to the transport level much of the need for message sequencing and completion signals employed in systems such as MSG [162]. Moreover, indivisibility further encourages a separation of the semantics of an operation from its implementation. Each service can be specified to the outside world simply by giving the virtual operations it provides. A complete set of message primitives is given in Appendix A.

Process structure and message protocols are discussed in Chapter 6.

2.2.3.2 Interhost Communication -

Interhost communication in RIGITS is provided by processes called network servers. Each RIGITS machine has at least one network server that handles the flow of messages to and from other machines. The function of a network server is to act as a local representative or liaison for remote machines, and to use the resources of the local machine on their behalf.

Figure 2 shows the path taken by messages over a network. A message sent from a local process, PA, to a process, PB, on a remote host is diverted by its Kernel to the appropriate network server process. The local server is responsible for routing and reliable transmission to the corresponding network server on the remote host. The network server is also responsible for any data conversion within messages required between the machines involved. The remote network server, upon receipt of a message from PA, forwards the message to its final destination, PB. PA and PB remain unaware that the message was routed through the network servers.

In order to ensure transparent communication between hosts, remote addressing must be provided. Typically, a hierarchical addressing scheme is used where each process address specifies a network, host, and local process number within that host. The Kernel routes messages to the appropriate network server on the basis of the network field. This approach requires that the Kernel know the address of the network server associated with each network.

A more general approach, adopted in RIGITS, is to allocate a local alias for the remote process. The alias is allocated from the pool of available local process numbers. With each alias, the Kernel associates a local process to which outgoing messages will be routed. Typically, an alias is created in cooperation with a network server when a local process attempts to communicate with a remote process on that network. Aliases eliminate the need for the Kernel to know all possible network addresses (so that it can route outgoing messages to the appropriate server), increase the local process address space (by eliminating the

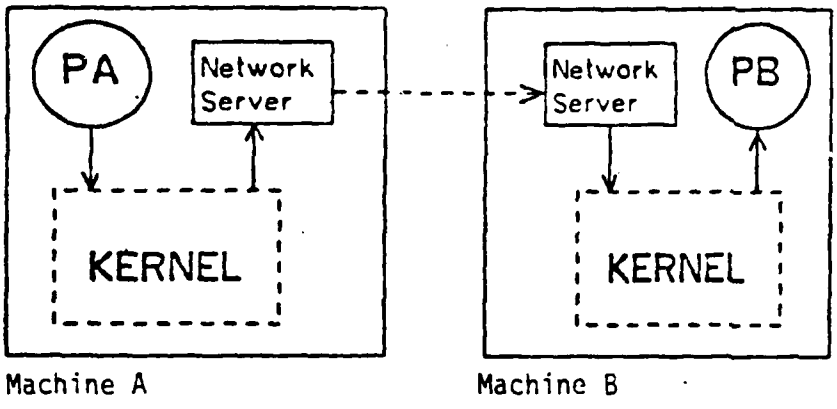


Figure 2. Inter-host communication.

need for multiple fixed-format fields), and provide the mechanism whereby any process can be interposed between any two communicating processes.

However, if an alias is embedded in an outgoing message, that alias must be translated into an address accessible by the remote process. This problem does not arise if global names are used. Address translation and related issues are discussed in, for example, [174, 190, 229].

2.2.3.3 Name Service -

A remaining question is how a process initiates communication with another (possibly remote) resource. RIGITS services are requested symbolically, such as a resource path name. Processes that are willing to provide services make themselves known to the rest of the system by registering with a name-server (or directory server) process. Subsequently, when the name-server receives a symbolic request for service, it can return the address of a process that provides the service (see Section 2.2.3.2). At the time a service is requested, the right of the requesting process to use that service can be verified, or a selection from a set of equivalent services can be made. If no process is currently willing to provide the service, one can be created (as in MSG).

A process making a name request may specify on which RIGITS network or machine the process should be found, but typically "broadcasts" the request. This allows the requesting process to ignore the location of the service. The local name server notifies each network server that it should broadcast the request on its associated network. The remote network servers communicate with their respective name servers to determine which host, if any, provides the service.

Usually, if the service is found on the local host it is accepted. If local service is unavailable and more than one host offers the service, all "bids" but the first are ignored. A more sophisticated Job Manager would consider the bids and use the most effective service.

Name service is necessary only at the initiation of communication. Thereafter, the specific address of each process is known to the other.

2.2.4 Exception Handling

Distributed programming imposes a heavy responsibility to handle a multitude of error conditions. Message activity can be pipelined or multiplexed, and the relationships between incoming and outgoing messages are much richer than in a conventional programming environment. A faulty process can conceivably crash many other processes by sending illegal messages, making it very hard to identify the process that caused the problem.

Because there is no shared memory, protection cannot be enforced by hardware. Because there is only one level of co-equal processes, protection is not enforced by layering. Instead, processes must first be designed with the principle of mutual suspicion firmly in mind [194]: Because the interfaces between processes are well-defined, it is possible for a process to validate the requests made of it and to check that other processes it calls upon have performed as expected. Connections can be used to provide a capability-like protection mechanism (see Sections 5.1.4 and 6.2.2).

In addition, RIGITS provides two fundamental mechanisms for handling exceptions. The term exception is chosen because, unlike the term "error," it does not imply that anything is wrong [84]. Intra-process exception handling is supported by a procedure-call-oriented mechanism that allows an error notification to propagate back up the "calls" hierarchy [165] to a designated point. Inter-process exception handling is supported by the use of asynchronous emergency messages. Emergency messages are delivered with highest priority — ahead of any other messages queued for the receiving process — and will cause a blocked process to be awakened. When an emergency message is received, the emergency handler associated with the process is invoked.

Emergency messages provide the means for more general event notification. The process registration facilities mentioned in Section 2.2.2 are supported by emergency messages. More general registration facilities are possible, whereby a process may notify an event handler that it wishes to be notified whenever a particular event, or combination of events, occurs. Notification is by emergency message.

Exception handling is discussed in Chapter 6.

2.3 The RIGITS User Interface

RIGITS attempts to provide the user with access to all available facilities through a single display terminal. In doing so, RIGITS presents a coherent interface that minimizes the confusion inherent in dealing with many different computers and tools simultaneously.

At any instant the user may be engaged in multiple, concurrent activities. The user must be able to have any number of his activities displayed on his terminal simultaneously (typewriter terminals are not supported). These activities all use Virtual Terminals to communicate with the user. Moreover, across all activities the command interaction discipline is made as consistent as possible.

The user interacts with three types of process:

- the Monitor
- Executives
- tools (application programs)

When the user enters RIGITS he is talking to the Monitor, which is responsible for managing the user's display terminal and maintaining state as to the user's activities. At the user's request, the Monitor allocates portions of the screen to particular jobs, groups jobs into screen images, and tells the user where everything is.

Each user job is associated with an Executive, which is the process to which the Monitor initially allocates regions of the screen. An Executive serves much the same function as a TOPS-10 monitor or TENEX executive (fork). Each Executive can perform at most one job at a time. The user may, however, have as many Executives (and hence jobs) running simultaneously as he desires. At his request, any combination of Executives may be mapped to the screen.

Some jobs, such as file management functions, are performed by the Executive itself. Other jobs involve the instantiation of a tool, consisting of a Tool Interface Process and one or more Tool Service Processes. The Tool Service Processes provide the requisite processing function, while the Tool Interface Process provides the interface to the user. Tool Service Processes and Tool Interface Processes are separated primarily to guarantee fast response to user input when the tool resides on a remote host.

Thus, the basic logical flow of RIGITS, to a single user, appears as follows:

1. When the user sits down at his terminal a Monitor is created for him. The Monitor processes commands for managing Executives and allocating screen space. If the "Quit" command is issued, all resources associated with the terminal are released. Assuming an Executive is given control -- via com-

mands or screen management keys -- the screen is remapped to contain an instance of the Executive and...

2. The Executive processes commands for file management and the like. If the "Quit" command is issued, the Executive disappears and the screen is remapped to activate another Executive or the Monitor. If a request is made, for example, to edit, or talk to the DEC-10, a tool is spawned. The tool will run in the same window allocated to the Executive.

3. The tool executes: The Tool Interface Process interacts with the user as necessary, and dispatches requests for services to the Tool Service Process(es). Upon satisfactory completion, the tool dies, whereupon control is returned to (2).

4. At any time the user may turn his attention to a different tool, Executive, or Monitor. (The Monitor may be thought of as a higher-level Executive, and Executives as special-purpose tools!)

2.3.1 Virtual Terminals

A Virtual Terminal is roughly equivalent to an independent physical display device. Each Virtual Terminal may be associated with an area of the user's display. The Virtual Terminals associated with dormant activities may occupy only a small amount of screen space (or none at all), while those of current interest may occupy a large amount of the display. The contents of a Virtual Terminal are in no way affected by the amount of screen space allocated to it. If a dormant process becomes active and the user requests that a larger area be used to display its Virtual Terminals, the effect perceived by the user is that previously invisible text becomes visible. Although all Virtual Terminals may accept output simultaneously, only one Virtual Terminal can accept user input at a time. Special keys and commands permit the user to switch his attention from one Virtual Terminal to another. (In the sequel, key means any distinct signal from any input device — e.g. keyboard, mouse, or chordset.) The control functions and layout for the RIGITS keyboard are given in Appendix B.

At any time a user may:

1. suspend or discard output to a Virtual Terminal;

2. abort or suspend the process associated with a Virtual Terminal;

3. scroll back and forth within a Virtual Terminal to review previous material;

4. specify a file or select previous text as current input;

5. rearrange his display by moving or changing the size of a Virtual Terminal.

At all times the user has control over what he sees and how he sees it.

For example, consider the typical program development cycle of editing, compiling, and loading. Figure 3 presents an image of the RIG screen editor: The editor has four Virtual Terminals, one each for the banner, status, command interaction, and text-editing. The "_" cursor identifies the Virtual Terminal expecting input. The user may switch between the command and text Virtual Terminals via the CHANGEVIEWPORT key (see Section 3.2).

```
Exec02/edit/Editor                    Directory:<200>kal
Locate:                                        File: <200>kal:novarc.sal
Change:                                        Mode: Insert

Command?
require "ALL:CPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 NewItems;
require 2000 PNames;


   ! Tuning parameters;

define  INPUT!BUFFER!SIZE       = 256,
        HEADER!FILL!CHARACTER   = "55,
        PAGE!LENGTH             = 58,
        PAGE!WIDTH              = 132,
        COMMAND!PROMPT          = [ CRLF & "Command? "];

   ! Dump file info;
```

Figure 3. Editing a file.

In order to compile a program and correct errors while the compilation is in progress, it is useful to allocate one area of the screen to an editor and one to the compiler. Compilation is halted at each error, the necessary correction is made in the file, and compilation proceeds. Figure 4 presents this arrangement: In addition to the four Virtual Terminal associated with the editor, two Virtual Terminals for the SAIL compiler (running on the DEC-10) are shown, one for the banner and one for user interaction. The user directs his attention to the appropriate job via the CHANGEREGION key. Although the editor's Virtual Terminals are mapped to the entire screen in Figure 3 and only a portion of the screen in Figure 4, this has no influence on any of the properties of the Virtual Terminals.

```
Exec03/compile/TenTelnet                Directory:<200>SYS
  TTYUTS.HDR[170,166] 1
  SCNUTS.HDR[170,166] 1
  GPUTS.HDR[170,166] 1
  DATES.HDR 1
INSERTING FORGOTTEN SEMI-COLON.
TEST.  PAGE 1
04000       require 2000 NewItems;

^
Exec02/edit/Editor                      Directory:<200>kal
Locate:                                          File: <200>kal:novarc.sal
Change:                                          Mode: Insert

Command?
require "ALL:SCNUTS.HDR[170,166]" source!file;
require "ALL:GPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 New!Items;
require 2000 PNames;
```

Figure 4. Two activities sharing the same terminal.

Virtual Terminals are device-independent. If an application program needs to be aware of the particular device through which it is communicating, the program can be provided with the Terminal Profile for that device. The Terminal Profile defines, for example, the dimensions of the display and the control functions provided.

Virtual Terminals are the subject of Chapter 3.

2.3.2 Command Interaction

The user would like to interact with his various activities in a coherent fashion. Across all jobs he should be able to type the same control keys and issue similar commands to get help, rearrange his screen, and the like. Moreover, he should be able to tailor the user interface to his own preferences -- disable prompting or redefine control keys, for example. User tailoring is provided by User Profiles.

The command interaction discipline is discussed in Chapter 4.

2.4 Profiles

To provide dynamic configurability, RIGITS relies heavily on declarative profiles. Profiles are associated with entities such as terminals, hosts, tools, processes, commands, and users. A process that deals with such entities is written in terms of a prototype whose characteristics may be bound at run-time by a profile.

This principle is most apparent in the use of Virtual Terminals: Application processes deal only in terms of Virtual Terminals. The Virtual Terminal Management System, which sits between the user and the application program, must provide different input and output facilities corresponding to the capabilities of the user's terminals. The Terminal Profile provides the necessary information (see Section 3.3.3).

Similarly, the User Profile defines the access rights of the user (for the Job Manager) and his desired command interaction environment. Command Profiles define the characteristics -- keywords, parameters, help text, and executable code -- for individual commands. Process Profiles are used by the Process Manager to create processes. Host Profiles specify the services (resources and tools) provided by particular hosts, and login/out sequences for network access purposes.

Profiles may be represented in a general, variable-field, syntactic format, namely, the PLITS message. Tools must be provided for creating and editing profiles, and, possibly, for "compiling" them into more machine-amenable forms.

2.5 Historical Perspective

The RIGITS message-passing paradigm grew out of experience with interprocess communication facilities developed for the Stanford Hand-Eye Project [75] and the work of Walden [221]. Systems that bear a resemblance to RIGITS include DEMOS, Thoth, RSEXEC, NSW, DCS, DCN, MININET, Roscoe, StarOS, Medusa, and LLLNOS.

DEMOS and Thoth are single-processor, message-passing systems. DEMOS provides rudimentary process registration and name service facilities, as well as capability-like access control to communication paths [17]. A link is created by the task (process) to which it points and then passed to the potential sender task. The creator may specify attributes of the link in order to identify incoming messages and to protect against unexpected or unauthorized messages. Links are used in conjunction with channels (RIGITS ports). Data segments may be associated with links such that memory can be shared between tasks; this provides the primary tool for communicating blocks of data.

Thoth was designed to be portable [39]. It emphasizes the use of many inexpensive processes, efficient process addressing, inexpensive interprocess communication, and dynamic configurability. Data sharing is effected via teams of processes (similar to Extended-CLU guardians [136] and the task forces of StarOS and Medusa); teams communicate via messages. Message-passing is fully synchronized; a message is not sent until a request for the message is outstanding. Services are typically provided by remote procedure call, and processes cannot block on the availability of resources. Thoth distinguishes between system processes (which access kernel services via system calls) and user processes (which access system processes via message passing).

RSEXEC was an early network operating system designed to provide access to a collection of TENEX hosts [213]. It is implemented in a manner similar to the RIGITS network servers: Requests for remote access are directed at a server process (RSSER) on the appropriate host. A distributed file system allows uniform accessibility via a file-naming syntax that has simply been extended to include a host field. Convenient access to frequently used files is provided by partial pathnames that are interpreted in the context of the user's "working directory." The working directory spans host boundaries and includes those directories the user normally access. RSEXEC is completely distributed in concept and implementation, and the distribution is visible to the user. Distributed file storage with the possibility of multiple copies gives rise to problems of consistency and protection that are still unsolved.

The National Software Works is fundamentally a system for software production [96]. It is intended to provide programmers with uniform access to a wide variety of software production aids, and managers with access to a collection of management tools for monitoring and controlling project activities. It is implemented as a collection of processes for user interaction, resource management, and file movement and translation, built on top of a fairly sophisticated interprocess communication facility, MSG [162]. In comparison to the uniform RIGITS message interface, MSG provides three primitive modes of interprocess communication — messages, connections, and alarms — together with asynchronous mechanisms for signaling message reception and the ability to dictate the sequencing of messages. The distribution of the NSW is totally invisible to the user; neither the syntax nor the semantics associated with the user interface includes any provision to specify actions directly relating to the distributed nature of the system.

DCS was one of the first and best distributed systems [67]. Processes distributed across a (ring) network communicate by messages. Messages are sent to ports, several of which may be owned by a process. Communication is initiated by name, and the physical location of a process is irrelevant. The file system is distributed among all the machines, and is designed for fail-soft performance. The ring architecture allows resource allocation based on bidding: Messages may be broadcast to every processor soliciting bids for a desired resource. This feature, however, was never implemented.

DCN is basically DCS with a point-to-point communication network rather than a ring [150]. MININET consists of a packet-switched communications subnetwork interconnecting message-switched host operating systems [145]. Inter- and intra-host communication meet the goals of uniformity, but communication is transaction-based, the hosts are homogeneous, and the entire system is implemented directly on the hardware.

Roscoe is a RIGITS-like system for a physically and functionally homogeneous, close network of LSI-11's [200]. All traditional operating system functions are performed by utility processes (RIGITS servers). All communication among processes takes the form of messages across links (as in DEMOS). Roscoe supports a hierarchical, distributed file

system similar to that of DCS. Global resource management is achieved via an interconnected set of resource managers (RIGITS Job Managers), one per host.

StarOS is a message-based, object-oriented operating system for Cm* [112]. It was specifically designed to support task forces, large collections of concurrently executing, small processes that cooperate to perform a single task. All information in StarOS is encoded and stored in typed objects, which are accessed via capabilities. A process never suspends execution as a side-effect of a message send or receive. The TASK programming language [113] is used to construct task forces. Various dependence relations between the processes in a task force are used for process suspension and abnormal termination, and for responsibility chaining.

Medusa is another operating system for Cm* [163]. It too relies on the use of task forces containing many concurrent, cooperating activities. Activities within a task force may share memory and other objects, while communication between task forces is achieved only through messages. The sharing of control within a task force has led to the notion of a buddy, by means of which one activity may handle an exception on behalf of another activity in its task force.

LLLNOS is one of the most complete designs for a distributed operating system [230]. As befits a recent design, it contains most of the best features of other major systems and has made a serious attempt to formalize a distributed system architecture. Processes communicates solely via messages; resources are associated with server processes; name service is provided. Protection is based on capabilities and a partitioning of the network into domains of trust; systems within a domain cannot pose as sources of messages from systems in another domain. The protocol structure is layered and transaction-oriented (with provisions for sessions or connections). The service support layer, in particular, provides uniform mechanisms for resource naming and protection, data translation, message formatting, and sessions [77]. Although Watson was a principal contributor to the NLS system [226], LLLNOS has not yet seriously addressed the principles of user interfaces for distributed systems. Some of the ideas in LLLNOS have been implemented in NLTSS, a single-processor operating system for the CRAY-1 [61].

Lastly, the ANSI/ISO reference model for open systems architecture is designed to enable structured dialogues to be established, maintained, and terminated reliably between any two processes in any two workstations located in any two open systems anywhere in the world [58, 236]. Each workstation is a cluster of activities or processes, each able to perform a defined set of functions according to the set of procedures established for the workstation when it was defined. When two workstations wish to communicate they must first establish a message path or session through intervening communication networks. The processes may then exchange messages in agreed-upon languages (presentation formats) according to established protocols. The ISO standard provide 7 layers: physical control, link control, network control, transport end-to-end control, session control, presentation control, and

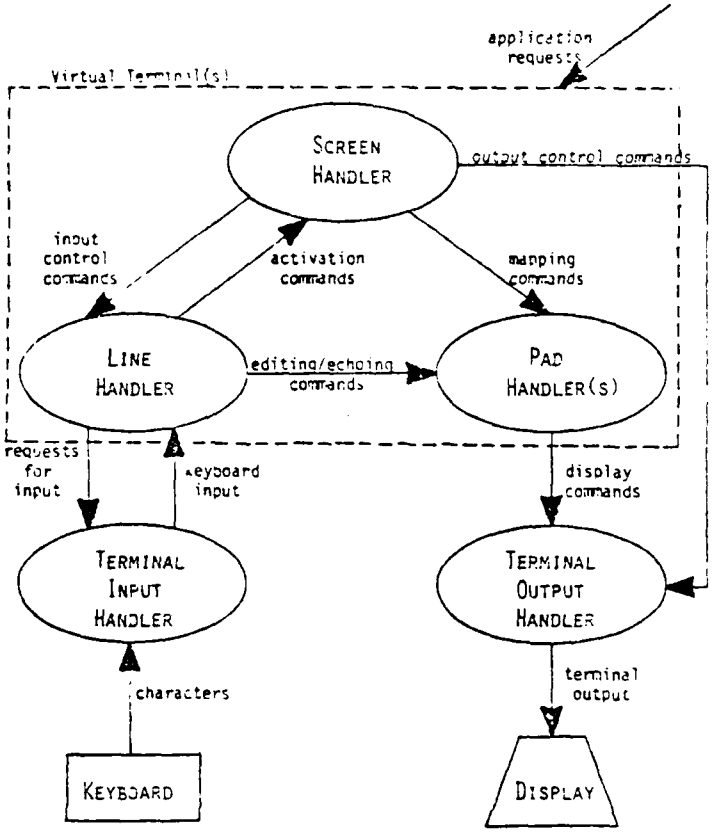


Figure 9. The Virtual Terminal Controller

editor process. If, for instance, a "?" was struck, indicating that the user required assistance, the editor process would then issue commands to the Pad Handler to output various help information. If a screen management key is struck, the Line Handler communicates with the Screen Handler to activate a different Virtual Terminal. Subsequent input will be interpreted in the context of the new Virtual Terminal.

5. Eventually the editor process makes another request for input and control returns to step 1.

The division of effort within a VTC is based on a desire to maintain functional modularity and to distribute components among various processors. Using the Virtual Terminal Protocol, application processes deal uniformly with Screen, Line, and Pad Handlers, which communicate in turn with device-specific Terminal Input and Output Handlers. The protocol between Terminal Input and Output Handlers and the terminal varies with the terminal. For example, the initial version of

terminal at run-time on the basis of the Terminal Profile. This leads to reliability, flexibility, and maintainability across a potentially wide range of devices.

3.4.1 An Implementation Approach

In RIG the VTC is implemented as a collection of five or more cooperating processes. Each RIG terminal has its own Terminal Input and Output Handlers, created at system generation time. When a user first accesses RIG, a Screen Handler and a Line Handler are created. The Screen Handler is responsible for managing screen space (i.e., Windows). The Line Handler satisfies all subsequent user input requirements by communicating with the Terminal Input Handler and directing characters to the user's various Lines. Typically, one Pad Handler is associated with each Virtual Terminal's Pad and satisfies that Virtual Terminal's output requirements by sending display commands to the Terminal Output Handler.

Figure 9 diagrams the flow of information within a VTC. The Terminal Input Handler collects input characters from the keyboard, and passes them on to the Line Handler for interpretation. To echo a character, the Line Handler passes it to the appropriate Pad Handler, which stores the character in a Pad and passes it to the Terminal Output Handler for echoing on the display. Editing keys result in appropriate commands being sent to a Pad Handler, which performs the edit on a Pad and issues the appropriate display updates to the Terminal Output Handler. If a character is typed that causes a new Virtual Terminal to be activated, the Line Handler notifies the Screen Handler; the Screen Handler in turn notifies the appropriate Pad Handlers to start/stop mapping their output to the display, and tells the Terminal Output Handler which Pad is in control of the input cursor. Application processes make requests to the Line Handler for input, to Pad Handlers for output, and to the Screen Handler for formatting. The various components are, however, transparent to the application process, which regards them collectively as one or more Virtual Terminals.

In Figure 5 consider the case where the user was typing the "open" command to the editor. The input loop proceeds as follows:

1. The Input Handler receives the "o" and passes it to the Line Handler.

2. The Line Handler sees that "o" is not a break character and passes it to the appropriate Pad Handler.

3. The Pad Handler updates its Pad and passes the "o" on to the Output Handler for echoing.

4. Steps 1-3 occur for "p", "e", and "n". When an appropriate break character (e.g., carriage return) is input, the Line Handler recognizes this and returns the collected input to the

alert (ring bell)
clear screen
clear to end of line
delete character
delete line
insert character
insert line
move cursor
(over)write character
(over)write line

3.3.3 Terminal Profiles

A Terminal Profile is constructed by the VTC when the user first accesses RIGITS. The Terminal Profile contains, for example, the dimensions of the display and its contrast characteristics. It also contains a "map" of signals generated by the keyboard into the Virtual Terminal control functions that they represent — octal 10, for example, is mapped to the function DELETECHARLEFT. Each character may have at most one control function; a control function may, however, be generated by more than one character. Characters not mapped are given no special interpretation by the VTC. An application program must not rely on a particular signal (e.g. 8-bit code) representing a particular control function; that is, signals have no pre-assigned logical function.

The Terminal Profile is provided, at run-time, to application programs that wish to tailor their actions on the basis of the physical device through which they are communicating with the user. It serves a function similar to the "terminal mode word" features of systems like TENEX.

3.4 The Virtual Terminal Controller

Virtual Terminals isolate users and application programs from the characteristics of particular physical terminals. The set of Virtual Terminals associated with a particular physical terminal (or user) is managed by the Virtual Terminal Controller for that terminal.

Users and application programs are not concerned with the internal structure of the VTC. Users need only concern themselves with the concept of a Virtual Terminal that provides them with uniform access to input and output devices such as keyboards and displays.

Application programs communicate with the VTC via a Virtual Terminal Protocol (VTP). The VTP is invariant to the type of physical terminal through which the program is communicating. The protocol between the VTC and the terminal varies with the terminal. Where necessary, application programs may tailor their actions to a particular

```
require "ALL:GPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 NewItems;
require 2000 PNames;


   ! Tuning parameters:

define  INPUT!BUFFER!SIZE       = 256,
        HEADER!FILL!CHARACTER   = '55,
        PAGE!LENGTH             = 58,
        PAGE!WIDTH              = 132,
        COMMAND!PROMPT          = [ CRLF & "Command? "];

   ! Dump file info:

define  NAME!BLOCK!MARKER       = '377,
        DATA!BLOCK!MARKER       = '376,
        ERROR!BLOCK!MARKER      = '375,
        END!BLOCK!MARKER        = '374,
        TIME!BLOCK!MARKER       = '373,
```

Figure 8. Another Configuration.

3.3.2 Output

A display is considered to be any device on which some bounded number of text-lines may be shown simultaneously. A display may provide many different contrast or highlight characteristics, such as color or reverse video, intensity, blinking, underlining. Each characteristic may be represented by a field in a "contrast" mask; thus, Virtual Terminals may specify any combination of characteristics, while a particular display employs those characteristics for which it is designed. As with graphical input devices, the integration of graphical or audio output modes into VTMS is an open research question. See [6, 105] for some valuable comments about the requirements for an effective display terminal.

The format of VTC display commands has evolved out of a desire to minimize the amount of state information maintained by the VTC, and to make them terminal-transparent, that is, independent of any particular physical terminal. The commands attempt to incorporate the features provided by most (page mode) terminals, while leaving out some provided by more intelligent terminals. The commands include:

mand and text Virtual Terminals. Figure 7 shows an Image containing a single Region to which this Configuration of the editor is mapped.

```
Command'open (file) novarc.sai...open
Command?
Command??
Abort          Append         Change         Close          Copy
Exit           Locate         Move           Open           Pick
Quit           Read           Save           Transfer       Write

Command?_
```

```
require "ALL:CPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 NewItems;
require 2000 PNames;


   ! Tuning parameters;

define  INPUT!BUFFER!SIZE       = 256,
        HEADER!FILL!CHARACTER   = '55,
        PAGE!LENGTH             = 58,
```

Figure 7. An alternative Configuration.

On the other hand, the user may want all the available editor space devoted to text-editing. Yet another Configuration composed of the single text Virtual Terminal provides this (Figure 8). Successive Configurations are activated via the CHANGECONFIGURATION key. In sum, the editor manages four Virtual Terminals grouped into three different Configurations.

3.3 Physical Terminals

Lines and Pads represent logical keyboards and displays, respectively. Their physical counterparts constitute a terminal. Particular keyboards and displays are defined by a declarative profile available at run-time to application programs.

3.3.1 Input

A keyboard is considered to be any device capable of generating distinct signals in response to user input. Input devices such as mice, joysticks, or lightpens might be considered components of a keyboard in this generic classification, but their use presents several problems. Integration of such devices into VTMS is an open research question (see Section 8.1).

A Configuration is a description of the way in which a subset of a Superwindow's Windows (program's Virtual Terminals) should be displayed; it specifies the relative positions of the Windows, their relative sizes as a percentage of the whole, and actual viewing conditions. Each Window in the Configuration may have a size and contrast independent of the default size and contrast specified when the Virtual Terminal was created, although bounded by the Window limits. Processes may configure their Virtual Terminals in as many ways as they desire, but are not aware of which Configuration is currently active. The user changes Configurations via a special key.

Finally, because any realizable text display is limited in size, the entire context of the user's activities cannot always be visible on a single screen. For example, the user may wish to allocate the entire screen to a particular program while allowing other programs to continue in the "background;" the background programs may then be mapped to the screen at a later time. This facility may be provided in one of two ways: For raster displays, the capability of overlapping Regions can be provided. For typical display terminals, however, it is easier to use multiple screen Images. Literally, an Image is "what a screen might look like." It is composed of a set of Regions containing a subset of those Virtual Terminals associated with the user. The user may define any number of Images, swapping between them through the use of a special key.

In summary, for raster displays, Windows and Viewports suffice, but the remaining abstractions are recommended. Superwindows, Regions, and Configurations provide means of grouping the Virtual Terminals associated with application programs into logical entities. Images provide a fast way to "flip" the screen.

3.2.1 An Example

By way of further explanation, consider the following example: A user wishes to allocate half of a 25-line terminal to one program, say a DEC-TELNET, and the other half to an edit session. Through his Monitor the user creates an Image with two Regions — one for the Superwindow associated with the Telnet and the other for the Superwindow associated with the Editor. He then asks the Monitor to swap that new Image to his screen. The resultant display looks like Figure 4 (Chapter 2). In the top Region are two Viewports displaying respectively the banner and command Virtual Terminals of the Telnet process. In the lower edit Region there are four Viewports. The sizes of the Viewports (1, 9, 1, 3, 1, and 10 lines) are determined by the actual size of the Regions (10 and 15 lines) and the relative size information contained in the currently active Configurations of the associated Superwindows.

The default editor Configuration allows only one line for command interaction -- in order to maximize the space available for text-editing. If, however, the user needs assistance in specifying a command, he may require more lines for the command Virtual Terminal. This is provided by creating another Configuration containing only the com-

A Viewport is the 2-D area of the screen within which the contents of a Window are actually viewed. The size and video characteristics of a Viewport depend, in part, on the attributes defined for its associated Window. A Window may be mapped to more than one Viewport (on more than one screen -- see Section 3.1.2).

In the most general case the abstractions of Window and Viewport suffice. The user allocates Viewports on his screen and specifies which Virtual Terminal (i.e., Window) should be mapped to which Viewport. This may be the best approach for raster displays where Virtual Terminals can be used for menus and graphics, and can be overlapped quite easily.

Nevertheless, an application program often wishes to use several Virtual Terminals. From the user's viewpoint it is usually desirable to display all output pertinent to a particular program in a contiguous area of the screen. To ensure contiguous display all Windows associated with a program are logically grouped into a Superwindow, which may then be mapped to a Region (or Superviewport) of the screen. Only Windows associated with that Superwindow may be mapped to the Region. A Region, then, consists of a collection of contiguous Viewports. Just as a Window may be mapped to more than one Viewport, a SuperWindow may be mapped to more than one Region. Regions have fixed sizes and are created by the user. The VTC has the responsibility of allocating portions of the available Region to its constituent Viewports.

Given the additional abstractions of Superwindow and Region, it may now be sufficient for the user to allocate Regions on his screen and specify which Superwindow (collection of Virtual Terminals associated with a program) should be mapped to each Region. The application program could specify a fixed topology of Windows, that is, how its Virtual Terminals would be arranged in any Region.

However, it is not always desirable to observe all Virtual Terminals associated with a particular program. In the editor, for instance, it is useful to be able to eliminate the banner, command, and status Windows, and deal only with a (larger) text-editing Window. The most general solution to this problem is to provide the user with the capability of creating, rearranging, and deleting his Virtual Terminals at will; that is, we fall back to the simplest level of abstraction -- Windows and Viewports. Given the concept of inter-related Virtual Terminals that should be displayed together, an alternative approach is to have the VTC cycle through all

$$\sum_{i=1}^{n} \frac{n!}{(n-i)!}$$

possible permutations of n Virtual Terminals on user command. VTMS compromises the user to some extent by allowing the application program to specify what it considers to be a reasonable set of Configurations.

3.2 Managing Screen Space

A multiple process environment in which simultaneous activities compete for a user's attention presents a number of problems for a screen management system:

1. A physical display may not be large enough to accommodate all the information important to all concurrent activities.

2. The user must be able to organize his work visually so that related information is arranged logically on his screen.

3. A given program may wish to provide various viewing options (e.g., contrast, or "optimal" sizes) without wanting to complicate its internal state with elaborate screen state information.

VTMS solves these problems through a hierarchical decomposition of screen space reminiscent of the way computer graphics systems divide and map data onto graphics output devices (see [160]). There, portions of pictures termed windows are mapped onto areas of the display termed viewports. Similarly, in VTMS Windows on a Virtual Terminal's Pad are mapped onto Viewports of the screen. See Figure 6.

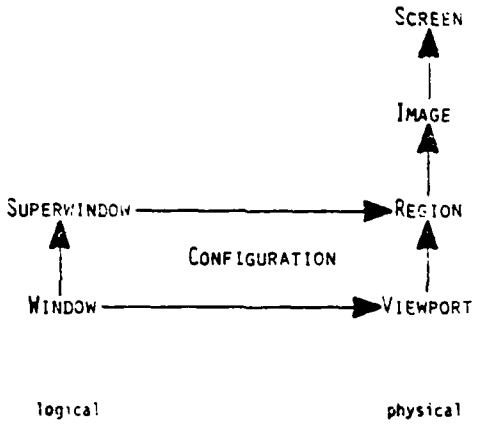


Figure 6. The dual hierarchy of screen primitives.

A Window represents a potential mapping onto a display of the output contained in a Pad. Its attributes include preferred contrast (such as inverse or blinking) and limits on its size when displayed. The Window definition does not specify where in the Pad it is. Rather, the Window is always implicitly located over the viewing cursor associated with its Pad (see Section 3.1.2).

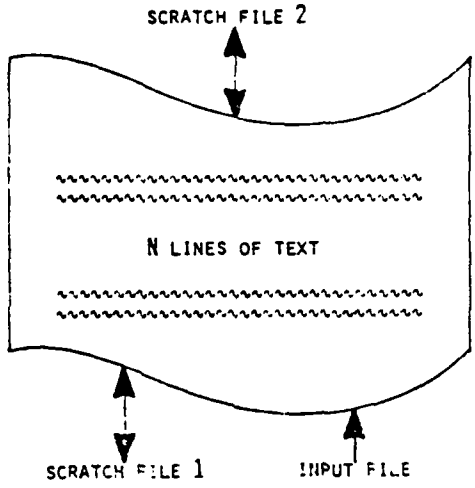


Figure 5. A Pad being used to edit a file.

A given Pad may be shared by more than one Virtual Terminal. This feature can be useful when a Pad contains status information updated by a single program but of interest to many system users. For example, the RIG "banner" process makes a Pad containing the RIG system version number, date, and time available to all users. Mapping the same pad to multiple Virtual Terminals also provides the means for two or more users to "link" or "share" screens — for teleconferencing and the like.

A Pad's output cursor is under the control of the program that owns it. On write operations its position is updated in a fashion analogous to that of a physical terminal. Normally, movement of a Pad's output cursor also changes the mapping of Pad lines onto the user's display, resulting in a scrolling action; that is, the "newest" data is always displayed. Alternatively, these mappings may be defined to follow a second type of cursor, the viewing cursor. There are as many viewing cursors as there are Virtual Terminals for a given Pad, each under the control of the user viewing that Virtual Terminal. Viewing cursors are usually linked to the Pad's single output cursor in such a way that movement of the output cursor also moves the viewing cursors. However, the user may detach a Virtual Terminal's viewing cursor for the purpose of reviewing past text and perhaps selecting it as input.

Output to a Pad may also be suspended or discarded. When suspended, the Pad refuses to handle any further requests from application programs until told to resume output by the user. In autoblock mode the Pad will automatically suspend whenever new text has filled the smallest area of any screen to which the Pad is mapped. (If the Pad is not mapped to a screen, it suspends output until it is mapped.) Special keys control suspension, discarding, and autoblock mode.

vious text. If that text has already been parsed (by a command interpreter, for instance), the results returned from the VTC must indicate that such text has been changed. The calling program may then reparse the complete text.

3.1.1.2 Indirect Input -

When processing commands it is often useful to specify indirect sources of input, such as programmable function keys or macro files. The processing of such input is distributed between the low-level keyboard driver, the VTC, and higher-level command interpreters. See Section 4.4.2 for details.

3.1.2 Virtual Output

The output capabilities of a Virtual Terminal are provided by an extensible data structure called a Pad. The pad represents the "store" of a Virtual Terminal.

Logically, a Pad is a cursor-addressable, two-dimensional, right-ragged array indexed by line number and character position. Each line may have a contrast type such as reverse video or blinking. The Pad maintains a fixed number of text lines in memory and uses two *scratch files (or an equivalent stable medium) for temporary storage.* In the event of a system crash, these files provide the means for recovering all but a Pad's most recent contents. The maximum amount of text storable in a Pad is therefore determined by the maximum file size on a particular RIGITS system.

VTMS provides a range of text-editing features through the use of Pads. These features may be initiated either by the application program or by the user and include:

- cursor motion by characters, words, lines, or pages
- deletion of characters, words, lines, or pages
- joining and splitting lines
- character overwrite or insertion
- string location and substitution
- text selection and transfer (copy)

Further, the contents of any RIGITS text file may be inserted into a Pad, or some or all of a Pad may be copied into a file at any time. Together with the ability to select arbitrary portions of text, these features allow the Pad to be used both for the editing of files (see Figure 5) and for the management of temporary text buffers. The latter facility provides the ability to use the output of one Virtual Terminal as input to another (similar to UNIX pipes).

correct Virtual Terminal.

The user may type a control key at any time. Control functions are related to particular keys via a lookup table, allowing different character codes to be used with different types of keyboards (see Section 3.3.3). This also permits the user to specify control functions himself, perhaps through his User Profile (see Section 4.4.1). Control functions are identical across all application programs.

The actions associated with control characters may be circumvented by including them in a set of break characters either at the time the Line is created or on any request for input (see the following section). Alternatively, any character prefaced with a PASS character will be treated as a normal character to be queued for the line in question.

3.1.1.1 Input Modes -

The program owning a Virtual Terminal may request that input be collected in one of three modes: In character-at-a-time mode a single character is returned in response to each request. Echoing is optional.

In page-edit mode characters typed by the user are allowed to modify the contents of the Virtual Terminal until a program-specified break character is typed. A program may also specify the set of "acceptable" characters such that any characters not in that set will be ignored. Page-edit mode is used primarily for editing files; indeed, it is simply a driver for the editing facilities of the VTC (see Section 3.1.2). All editing takes place within the VTC; the requesting program is not aware of any input/output until a break-character is typed.

Line-edit mode is used primarily for processing commands. All of the intra-line editing facilities of the Pad are available. Line-editing continues until a break character is typed. Entire text lines or single tokens may be line-edited. A text-line logically consists of three parts: 1) prompt; 2) previous text; and 3) current input (field). This is similar to the partitioning in TOPS-20 (COMND and TEXTI JSYSes). For example, if a command is being typed, previously parsed fields constitute the previous text; the current field constitutes the current input; and the prompt is the command prompt. Hence, the following partitioning may result (user input in upper-case):

```
Command? COPY  OLDFILE=TEMP1  NEWFILE=

|prompt |          text              | input
```

The user may inquire about the current input field at any time, e.g., ask for all options for which the current input is a prefix (see Section 4.1.5). His inquiry will be fielded by a process external to the VTC. It is therefore possible to pass partial input back and forth to the VTC, editing it as necessary. It may also be possible to edit the pre-

any number of Virtual Terminals, thus allowing it to manage different kinds of output separately. A Virtual Terminal may be written to or queried for user input in much the way a physical terminal can be used in a single-job-per-terminal system. In addition, a Virtual Terminal provides its owner with extensive facilities for editing text and the ability to save all output in data structures on stable storage (such as disk).

The Virtual Terminals associated with a particular user (i.e., display terminal) are all managed by a single Virtual Terminal Controller (VTC). The implementation of the VTC is transparent both to the user and to application programs.

3.1 The Virtual Terminal

A Virtual Terminal consists of three logical components:

1. Line - A Line serves as the Virtual Terminal's source of input.
2. Pad - A Pad is a stable-storage-based data structure used for storing and editing Virtual Terminal output.
3. Window - A Window is a potential mapping of a Virtual Terminal's Pad onto a display.

Lines and Pads will be discussed in the following sub-sections. Windows are the subject of Section 3.2.

3.1.1 Virtual Input

When multiple simultaneous activities share a single physical terminal, input from and output to that terminal must be multiplexed. Output can be multiplexed in both space and time by using the two-dimensional features of a display terminal. Input can be multiplexed in time only, through the use of virtual input devices termed Lines.

Any number of Lines may be created in the course of a RIGITS session, but only one Line may be "active" at a time, that is, receiving characters from the user's keyboard (or other input device). A Line is activated in response to some action by the user, such as typing a special function key or issuing a command. From the moment the Line is activated, subsequent user input is directed to it. Whenever a Virtual Terminal's Line is active a cursor appears in the associated Window to distinguish it from all others. Characters are echoed only when they are extracted from the input queue in response to a program request for input. This prevents characters from appearing in the wrong place on the user's display by ensuring that type-ahead always goes to the

CHAPTER 3

Virtual Terminal Management

RIGITS gives its users the freedom to perform any number of activities simultaneously. The management of these activities is made possible by the Virtual Terminal Management System (VTMS) [127, 128, 129]. VTMS allows a user sitting in front of a display terminal to view the output of various application programs on different areas of his screen. It provides him with commands to rearrange his display, edit and/or save its contents, and direct input to any of the programs under his control.

The evolution of VTMS from a vague notion of "multiple windows" into a working system was directed by four key beliefs:

1. The user must have complete, preemptive control of his terminal at all times. He should be able to allocate and arrange the space on his display device at will, selecting which Virtual Terminals to view at any one time. He should be able to tailor what he sees to his own preferences and needs.

2. Processes should never depend on the actual mapping of their output onto the user's display. A program may stipulate preferred viewing conditions, but these are applicable only as long as they do not interfere with the user's control of his terminal.

3. The output of a program should not be thrown away unless specifically requested by the user. The user should at any time be able to examine the past activity of his programs, possibly forming new input from data displayed on the screen. He should be able to save the output of any program as a transcript file.

4. Related output should be kept together on the user's display.

The fundamental element of the VTMS design is a powerful programming abstraction called a Virtual Terminal. Virtual Terminals permit application programs to remain unaware of the specific physical device through which they are communicating. An application program may own

applications. The work described in the thesis spans levels 5 through 7.

Begun in 1974, the basic RIGITS distributed system architecture pre-dates most of the above systems (the most noteworthy exception being DCS), and has been applied to a system that has been in everyday use since the summer of 1977. Although interprocess communication was well understood when the initial design for RIGITS was formulated (and had been implemented in a number of major operating systems -- RC4000, Cal, Elf, Hydra, TOPS-10, TENEX, B6700 MCP), such a total dependence on message-passing was a considerable deviation from the norm. Moreover, none of the above systems (with the possible exception of the National Software Works) has placed as much emphasis on the user interface as RIGITS. Overall, RIG(ITS) provides more facilities in a substantially more coherent fashion than other major distributed systems.

VTMS for the Altos consisted of Terminal Input and Output Handlers running on the Altos, and the remaining components of the VTC running on an Eclipse (see Section 8.1).

When a new terminal is brought into the system, a Terminal Input Handler and Terminal Output Handler must be provided for it. The remaining components of the VTC tailor their actions, at run-time, on the basis of the resulting Terminal Profiles, and need not be changed.

Moreover, the base functions of the Input and Output Handlers do not change. For example, all Terminal Output Handlers must handle line deletion and insertion; only the specific control signals for the terminal at hand must differ. Thus, the source code consists of a general handler that is combined with a terminal-specific set of support routines.

3.5 Historical Perspective

Development of VTMS began in the fall of 1975. Inspired primarily by Swinehart's work on debugging [207], it was an attempt to provide a much more versatile user interface than that available in any existing time-sharing system.

TENEX and TOPS-20, for example, provide for the creation of multiple executives, each of which may execute an independent program. However, the executives are stacked such that all lower-level executives must be killed in order to return control to a higher-level executive. UNIX provides the facility for "forking" multiple concurrent processes, but all processes expecting user input must type prompts to the same output stream, resulting in an indecipherable mixture of echoed input and program output. (The same characteristic is true of most systems, including TOPS-20 and TENEX, which provide the capability of running multiple processes from a single job.)

To correct these deficiencies, many systems provide a top-level control process that spawns a tree of independent sub-processes (executives) that do the actual work. Any given sub-process may be killed without affecting the operation of the others. In systems such as ITS and the National Software Works a form of "tty-passing" is implemented whereby the user's terminal always belongs to one process; this avoids the problem of mixed output from multiple processes. However, it is difficult for the user to remember the state of his various programs, and it is difficult for the system to provide the user with coherent information about important program activities.

Many of these difficulties (or features) arise from reliance on "typewriter" terminals. By taking advantage of the two-dimensional nature of available display terminals it is possible to allocate separate "windows" on the screen to different programs. This not only permits a user to view the output from many programs in a comprehensible fashion, but also allows him to suspend an operation, perform other operations, and then return without loss of context. This facility has

been incorporated into several computer systems — e.g., NLS/Augment, ZONES, POCCNET, TTDL, the TSO Job Session Manager, and Virtual Terminal UNIX — but has more typically been restricted to applications subsystems such as Copilot, Smalltalk, DLISP, SIGMA, and Nexus [74].

Compared to VTMS, few of these systems are as deeply ingrained in the infrastructure of the associated computer system, or offer an equivalent range of function. The fundamental difference is that VTMS concentrates on providing a complete Virtual Terminal in every window on the screen, whereas most other systems restrict each window to a specific sub-task. Few systems provide the extensive line- and page-editing features, or the ability to save terminal output on stable storage. Lastly, the techniques for screen management (beyond the concepts of Windows and Viewports) are unique to VTMS.

CHAPTER 4

Tools and the Command Interface

The Virtual Terminal Management System described in Chapter 3 becomes most useful when employed in the execution of application programs, or tools. These tools may be located on any host accessible through RIGITS. In order to ensure sufficiently rapid response the user interface processes should be as close to the user as possible, while the tool service routines possibly reside on a remote host. This leads to a logical and physical separation between tool interface processes and tool service processes.

Moreover, the user should be able to interact with any tool in a consistent manner. Because RIGITS does not intend to integrate all available hosts into its process environment, many tools will maintain tool- and host-specific styles of interaction with the user (within the context of a particular Virtual Terminal). However, wherever possible the user should be buffered from the idiosyncrasies of each tool. The guiding assumptions are:

1. The command language should be transparent. The user should not need to be aware of any system peculiarities, but should be able to capitalize on those peculiarities if desired. In the latter case the language should permit invocation of an arbitrary, named service implemented by a local or remote resource.

2. The command language should be simple and uniform. The commands should perform the actions their names imply, and should be simple in format and machine-independent. Default parameters should be supplied and an arbitrary number of parameters should be allowed.

3. The interface must provide enhanced, consistent "help" and error recovery facilities. Standard mechanisms should be provided for presenting status or error conditions to the user, independent of the host generating the error.

4. The interface should be amenable to various grades of user proficiency.

5. The user must be able to tailor the interface to his own preferences, carrying on the tradition from VTMS. He must be able to write various types of command programs, from simple macros to command programming languages.

6. The command repertoire should be easily modified and extended. Builders of new interactive application programs must be provided with facilities for easily creating the user interfaces for their programs.

7. Application programs should be provided with well-formed commands. In general, once a command is dispatched, further interaction with the user should be unnecessary, although it is not prohibited.

8. The command interface must provide fast response to user interaction.

4.1 Principles of Command Interaction

A command interface should allow the user to "express his need with constructs that are similar to his thought processes, natural problem-solving vocabulary, and language forms" [226, p. 357]. Such an interface might include a textual command language, graphical input and output modes, menus, and any other input/output modalities suited to the task and the user.

4.1.1 Language Forms

Command languages take two forms: 1) command interpretation languages (or interactive command languages); and 2) command programming languages. I will not discuss command programming languages in any depth (see Section 4.4.2), but it is important to design command interpretation languages that may easily be extended to command programming languages (see, for example, [217]).

Typical command interpretation languages take one of three forms: 1) positional; 2) keyword; and 3) natural language (English-like). For example, assume the user is making a copy of a file. In a keyword language the user might type:

```
COPY OLDFILE=FILE1 NEWFILE=FILE2
```

OLDFILE and NEWFILE are keywords. FILE1 and FILE2 are the values of the keywords, i.e., two specific files from the class. Each (keyword, value) pair corresponds to an operand or parameter. The order in which

the parameters are given is of no consequence since their values immediately follow special keywords. Keyword languages are attractive for novice users since the user need not remember the order of operands.

The corresponding input in a positional language (e.g. TENEX) might be:

```
COPY FILE1 FILE2
```

Keywords do not appear. The source and destination files are understood from the order in which they are typed, and the order is paramount! Positional notation is preferred by experienced users (and quick typists), yet can prove dangerous — consider the obvious case where the user accidentally confuses the order of the files in the above command.

In an English-like language (such as one constructed using LIFER [90, 91]) the equivalent input might read:

```
PLEASE COPY FILE1 TO FILE2.
```

or:

```
PUT IN FILE2 A COPY OF FILE1.
```

The same command can be specified in many ways in an English-like language. Noise words must be recognized and ignored. Although English-like interfaces are attractive, the supporting grammars are frequently ambiguous for a number of well-known reasons (see [100]). For many applications this may be permissible, but ambiguity can be rather dangerous when issuing file management commands. Moreover, in current systems, what appears perfectly "natural" may have many equally natural equivalent expressions that the command interpreter can not recognize; this can be quite frustrating for the novice user.

4.1.2 The RIGITS Command Interpretation Language

The approach taken in RIGITS is a synthesis of the three forms. It is based on Treu's realization that, in spite of surface dissimilarities, it is possible to characterize all types of user interaction by means of "action primitives" -- basic elements and their interrelationships that must be conveyed [216]:

- action verb
- action qualifier(s)
- object(s) of the action
- object qualifier(s)

No matter what (meaningful) action is conceived in the user's mind, the nature and object(s) of the action as well as any associated qualifiers must be produced. Therefore, stripped of extraneous words and other embellishments, a command can be represented by means of a general, variable-field, syntactic format. This becomes important when thinking about how to pass commands back and forth between processes. In RIGITS, that format is provided ideally by the PLITS message structure [71]: Each action primitive or command parameter is represented as a (name, value) slot.

In keyword languages, the action verb corresponds to the command name, objects are specified by the (keyword, value) pairs, and qualifiers are given by global and local switches. In English, the actions and objects are explicit, and qualifiers are given by adverbs and prepositional phrases.

The RIGITS command interpretation language is an attempt to provide what Treu refers to as a "user-oriented artificial language." The prime consideration is power and flexibility within a construct that is easily understood (and in some instances reads like natural language), and amenable to various grades of user proficiency. The basic command syntax is:

```
<command>  ::= <operator> {<operand>}*
<operator> ::= <verb> [<noun>]
<operand>  ::= [ <keyword> = ] <value1>
<value1>   ::= <value> [ ({<operand>}*) ]
<verb>     ::= command action
<noun>     ::= <verb>-dependent qualifier
<keyword>  ::= <operator>-dependent object-specifier
<value>    ::= <keyword>-dependent object
```

Here <verb> is the action verb, possibly qualified by the <noun> and (keyword, value) pairs (representing global switches). Objects are specified by (keyword, value) pairs, possibly qualified by additional (keyword, value) pairs (representing local switches).

Operands may be specified as (keyword, value) pairs or positionally; if a keyword is not specified, the position of the value determines the operand based on the structured definition of the command (see Section 4.2.2). The user typically will not mix positional with keyword arguments. Experienced users will tend to use positional notation whereas novices find keyword notation more helpful.

Each parameter has many characteristics:

1. type

   Typically, the value of a parameter is typed -- e.g. filename, string, or integer. If the input value does not pass the type check, the user is informed of his error. A "literal" type is

provided in order to circumvent type-checking.

2. optionality and default

   A parameter may be optional, and therefore must have a default value. If no value is supplied for the parameter and a default is available, typing a termination character as the first input character will result in the default being accepted. The user can display the default without accepting it by asking the command interface to EXPAND a null string (see Section 4.1.5). Alternatively, the default may be specified as a "constant" value, such that the parameter will not be prompted for and can not be specified.

3. prompt and feedback (post-prompt)

   Each parameter may have a prompt associated with it which can assist the user in typing the value. Once the value is specified, a post-prompt may be issued as feedback.

4. lists

   A parameter may specify that a list of values is expected. Values are separated by a user-definable delimiter (e.g. ",").

5. wild-carding

   When the type of a parameter is a file (or other appropriate object), an additional indication should be made as to whether wild-carding is allowed. For example, "FOO*.*" means all filenames beginning with "FOO". (Services similar to TENEX GTJFN/GNJFN, TOPS-20 GTJFN, and UNIX inspired the sophisticated file iteration facilities currently provided in RIG.)

More than one word (external keyword) may be used to represent the same keyword. External keywords are user-definable and spelling correction is provided.

For example, the above COPY command might be specified in RIGITS as any of the following:

```
COPY FILE1 FILE2
COPY FILE1 NEW=FILE2
COPY NEWFILE=FILE2 OLDFILE=FILE2
```

depending on the definition of the COPY tool (see Section 4.2). If data transformation or a similar operation was desired, this would be indicated by qualified values:

```
COPY OLD=FILE1 NEW=FILE2(BLOCKSIZE=132)
```

4.1.3 Command Input

A command may be specified using a variety of methods. Textual input is the typical input mode. Menus can be used to display all options for possible command verbs, keywords associated with a particular command, or values possible for a particular parameter; the appropriate option is selected with, for example, a mouse or chordset. Alternatively, once the command verb is specified, a "form" for the command can be displayed and the appropriate parameters specified.

When typing any parameter — command verb, keyword, or value — that can take a fixed set of values, it is possible to expand (complete or recognize) an input as if it were a prefix of the value(s). If a delimiter is typed and the input is not a prefix of the values, the user is informed (via a bell, for instance) that further input is expected. If a special EXPAND key is typed the input is expanded to match the greatest common prefix of the values it matches (see Section 4.1.5).

Feedback is provided on a per-parameter basis. The user does not have to wait until the command is completed to discover that he mistyped the first parameter. When an error is encountered, the user has the option of correcting it. A command is <u>never</u> aborted except by explicit user request.

4.1.4 Context

Many of the problems with man-machine communication arise from the lack of machine-maintained context. People maintain dialogues and pursue goals, but often in command interpretation each command is considered independently of any previous user input. The machine simply does not grasp the intent of what the person is doing.

At the very least it is important to maintain a history-list for previous interactions. This provides a context for "redoing" and "using" previous commands (as in INTERLISP [211]).

4.1.5 Help

Much of the power of a command interface resides in its help facilities. At any time a user must be able to find out what he has done, is doing, and can do with respect to the parameter or command he is typing or the tool he is using -- that is, in parameter, command, or tool space. Assistance may easily be rendered during parameter or command specification by providing keyword completion and spelling correction, help features to remind the user of syntax, and

user-oriented diagnostics to facilitate identification of errors. Help facilities in tool space include semantic data bases, tutorials, and expert assistance. Ideally, all documentation should be accessible on-line, both as a complete tutorial and as small sections chose in arbitrary order.

In the context of Chapter 3, at least three special keys seem important:

1. The PROMPT key will always display an appropriate prompt followed by a list of options. If typing a command verb (in tool space) or keyword (in command space) the PROMPT key will result in a display of all commands or keywords matching the current input. If typing a value (in parameter space), either the available options are displayed, or, in cases where the value does not come from a fixed set, the syntax for the value is displayed.

2. The EXPAND key is used for completion and recognition. Whether typing a command verb, keyword, or value, EXPAND will always expand the current input to match the greatest common prefix of the available options.

3. The HELP key provides for semantic help in the appropriate space. This may involve display of BNF syntax, suggestions for proceeding, tutorials and the like.

In addition, all errors require precise description and suggestions for corrections. The simplest form of error detection is to check the user's input for spelling and syntactic errors. Logical errors, such as specification of conflicting parameters, should also be detected. Distinctions may be drawn between "warnings" (where the user may continue with the command as specified, or correct it) and "errors" (which require user intervention).

4.2 Tools

Each task the user performs is typically associated with an application program, or tool. A tool must perform two functions: 1) interact with the user to determine what needs doing; 2) perform the requisite function(s). These services may be physically as well as logically separated for four reasons:

1. In a distributed environment such as RIGITS the service routines may be located on a host quite distant from the user. In order to guarantee fast response to user input, however, the interaction services should be as close to the user as possible.

2. The RIGITS environment does not provide for the encapsulation of all existing tools. It is therefore appropriate to access the computational services directly, while handling the user interaction through RIGITS facilities.

3. The tool service routines may be of use to more than one application program. For example, both the COPY and ARPA-FTP programs may want to access FTP services directly. This is possible by providing two different interaction modules which access the common FTP services.

4. Tailoring the interface to a particular user becomes much easier — he need simply supply his own interface routines or provide a different Command Profile (see Sections 4.2.2 and 4.4.1.)

In short, for any tool there is exactly one interface module, and one or more service modules.

4.2.1 Tool Interface Processes

A tool-specific Tool Interface Process (TIF) is responsible for interacting with the user and presenting a standard interface to the Tool Service Process(es) (TSP). By performing semantic checks on user input the TIF minimizes interaction with the possibly distant service. By handling responses from the TSP, it can present standard representations for errors rather than the obscure messages generated by traditional time-sharing systems.

One primary function of the TIF is looking for inconsistencies between parameters. For example, if the user is specifying arguments to a text-formatting system, the TIF should detect if the left margin is greater than the right margin before dispatching the request to the formatter. Specification of a particular set of parameters may necessitate that another (dependent) parameter be specified. In both instances the TIF requests the user to correct the command specification. Another approach is to define a language for validity checking and incorporate an appropriate "check definition" for each parameter in the Command Profile, such that the Command Interpreter could perform the checking (see Section 4.2.2).

A TIF must also handle responses from the TSP in an intelligent manner. The TIF itself can incorporate a Response Handler for the TSP in question, or the TIF could interface between the TSP and a general Response Handler. The Response Handler is responsible for providing a machine-independent RIGITS diagnostic to the TIF. Based on the context provided by the Response Handler and by the previously specified command, the TIF must be able to guide the user in recovering from errors. Rather than abort the request, the TIF determines those parameters that were in error, and requests their re-specification. It then re-dispatches the request to the TSP.

4.2.2 Command Profiles

It would be a frustrating and expensive task to replicate the command interaction discipline discussed in Section 4.1 in each TIF. In RIGITS the natural approach is to relegate much of this interaction to another process, a Command Interpreter. If, however, the Command Interpreter is to be TIF-specific, it must know the requirements of the TIF. This information is provided by Command Profiles. (As noted above, a particular command may result in multiple Tool Service Processes being invoked. Hence, the Command Profile does not describe a Tool Service Process, but rather the syntax of a particular command.)

A Command Profile describes the parameters to the associated tool (see Section 4.1.2) and specifies the program (TIF) to be run to process the command. In addition, the Command Profile specifies whether or not the command should be confirmed, whether literal parameters are allowed, and the like. A semantic help data base may also be referenced. The template for a Command Profile is given in Appendix C.

A consistent command interaction discipline is enforced by the Command Interpreter. For a given command, the Command Interpreter uses the associated Command Profile to control validity checking and prompting. Prompting includes the identification of parameters, defaults, and valid values, and can be requested at any time during command entry.

Each user may have his own set of Command Profiles for the tools at his disposal. Command Profiles may be altered either permanently or for the duration of the session by the user.

4.2.3 Summary

In summary, a tool is composed of:

- a Tool Interface Process
- Tool Service Process(es)
- a Command Profile
- optionally, a semantic help data base

4.3 Command Interpretation

Tool Interface Processes, Command Profiles, and Command Interpreters combine to perform the act of command interpretation. Each Command Interpreter has a fixed repertoire of commands, corresponding to the various tools (sub-tools, sub-sub-tools) available. Once a command verb has been recognized and authenticated (i.e., the user has authorization to use the associated tool), the Command Profile associated with that

tool is accessed. The Command Interpreter attempts to parse the command to completion before dispatching the command to the associated TIF. If the Command Profile indicates that particular slots are required, such slots must be filled, i.e., values specified by the user. Any optional slots may also be filled at this time.

When the command is completed, it is packaged into a PLITS-style message and given to the TIF. If the TIF or TSP notes that a semantic error has occurred, the TIF requests the Command Interpreter to interact with the user in order to correct or complete the command.

Command Interpreters interact with the user via Virtual Terminals. The Virtual Terminal Controller associated with the user's terminal provides the necessary input and output facilities. The Command Interpreter provides type checking, defaulting, prompting, and the like.

The appropriate Virtual Terminal may not be mapped to the screen when an error is encountered. When this occurs, the error message is normally posted to a Status Server as well as being displayed on the Virtual Terminal. The Status Server alerts the user to the error (via a bell, for example) and the user may respond accordingly.

It is important to remember that a tool may have subcommands. The TIF then has its own sub-Command Interpreter that interprets sub-commands on the basis of sub-Command Profiles. This hierarchy may extend indefinitely. In a RIGITS environment the most straightforward implementation approach is as follows:

1. The Executive starts its Command Interpreter. The Command Interpreter will take care of most user interaction; that is, TIFs will communicate with the Command Interpreter, which communicates with the Virtual Terminal Controller.

2. When the Executive Command Interpreter has recognized and parsed a command, it instantiates a TIF process for the tool (see Section 5.1.1), passing it the name of the Virtual Terminal through which it is communicating.

3. If the TIF has subcommands, it may instantiate its own Command Interpreter with the appropriate Command Profiles. That Command Interpreter is then given control and executes commands in the same way as the Executive Command Interpreter. The TIF can proceed with any other actions. In fact, if it has several types of interaction with the user (i.e., several Virtual Terminals), the TIF may spawn a Command Interpreter for each Virtual Terminal.

This approach results in the process structure shown in Figure 10. A given Command Interpreter is always one level above its tools in the hierarchy; that is, the Command Interpreter associated with the Executive manages the base-level tools, while the Command Interpreter associated with a tool manages sub-tools, and so on. The Executive may be

considered as a special-case tool with subcommands.

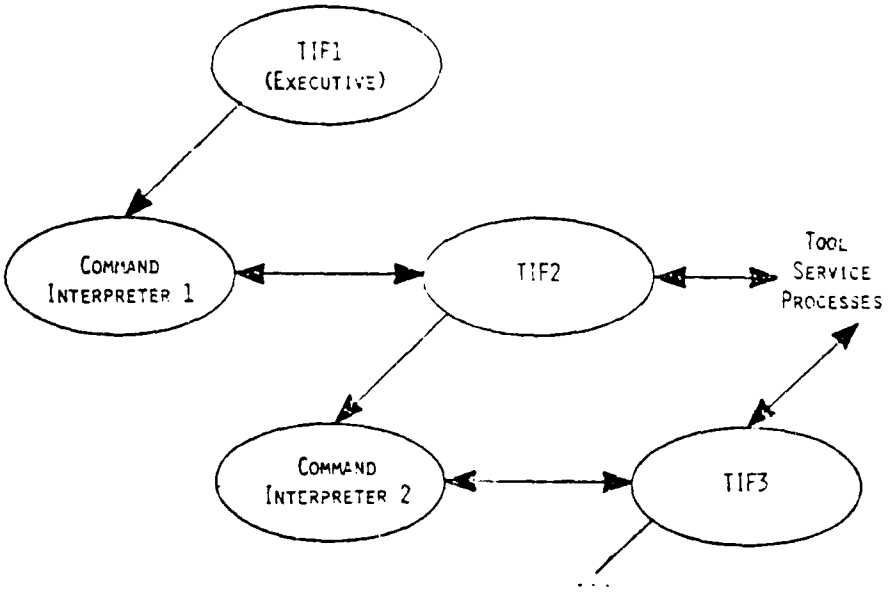


Figure 10. The command interface.

In more detail, a typical Command Interpreter might function as follows:

1. Get a command verb from the user and access the corresponding Command Profile.

2. Interpret the Command Profile, interacting with the user until either:

   a. he aborts the command, in which case goto (6.0); or

   b. he accepts the command, in which case goto (4.0); or

   c. the input must be checked against an authenticator, in which case proceed to (3.0).

3. Dispatch the request for authentication and wait for a response:

   a. success --> goto (2.0)

   b. failure --> display some help and interact with the user, perusing the help data base as in step (2.0). If the help session terminates successfully goto (2.0), otherwise goto

(6.0).

Note that authentication can proceed in parallel with further command interaction; that is, the request for authentication can be dispatched, the slot in question marked, and control returned to (2.0). If a negative answer comes back from the authenticator, the user will be notified when he next finishes a slot.

4. Spawn the appropriate TIF, giving it the command as an argument. Wait for a response from the TIF:

   a. COMMAND COMPLETED (start a new command) --> goto (5.0)

   b. COMPLETE COMMAND (the last specified command was not complete) --> goto (2.0)

   c. CORRECT COMMAND (a semantic error was detected) —> reset the current command state to the state indicated in the request and goto (2.0)

5. Store the command on the history-list.

6. Release all state associated with the command and goto (1.0).

4.4 Tailoring the Interface to the User

A fundamental goal of RIGITS is to allow the user to do anything he wants in the way that he wants. In Chapter 3 I described how the user may rearrange his display at will, organizing output from related tasks in ways that best suit his needs. He must also be able to affect the manner in which RIGITS interacts with him, and to write his own "executable" command programs.

4.4.1 User Profiles

The User Profile defines the user to RIGITS. It contains, for example, his passwords, access rights, mailing address, default directory, and a specification of his desired command environment. Access rights include his available machines, tools, devices, and files. The User Profile provides the means for automatically logging a user into remote systems.

The User Profile also contains a specification of the user's desired command environment. He may, for example, specify the degree of prompting and feedback, the control characters he wishes to use for particular functions, and programmable function keys. For each

In RIGITS, the emergency message provides the means for one process to alert another to the occurrence of an exceptional or unusual event. Emergency messages are implemented almost entirely with the same message-passing mechanisms as ordinary messages. By convention, an emergency port (-1, say) is reserved in each process for emergency messages. Messages delivered to that port are simply received with the highest priority, that is, delivery is independent of any outstanding messages to the receiving process. The only exceptions to normal message-passing are: 1) the queue for the emergency port is infinite, and thus a sending process will never be blocked when trying to send an emergency message; 2) the receiving process will receive the emergency message even if he is currently trying to send. Because a standard message is used, more data may be transmitted than just an interrupt signal. Moreover, because emergency messages are only delivered at "clean points" (when the process attempts to send or receive messages), the problems of synchronization between interrupt and data channels are simplified.

When a emergency message is received, the emergency handler associated with the process is invoked, and is responsible for processing the event. This leads us to a discussion of exception handling.

6.3 Exception Handling

Distributed programming imposes a heavy responsibility to handle a multitude of exceptions. Message activity can be pipelined or multiplexed, and the relationships between incoming and outgoing messages can be much richer than in a conventional programming environment (i.e., one in which subroutines are used as the primary structuring mechanism). A faulty process could conceivably crash many processes by sending illegal messages, making it very hard to identify the source of the problem. As a practical matter, it is difficult to ensure complete compatibility between similar programs written by different individuals in different languages. Supposedly interchangeable processes may differ in subtle, difficult-to-detect ways. Therefore, in order to provide an adequately robust program, the programmer of a distributed computation must give more thought to the problem of dealing with errors and exceptions.

Exceptions can be divided in two dimensions: 1) synchronous vs. asynchronous; 2) those that arise within a process and those that are external to the process. Typical intra-process exceptions include attempts to send incorrectly addressed messages or memory faults. Intra-process exceptions are invariably synchronous. Typical inter-process exceptions include the invalidation of previously sent requests or the death of a communicating party. Inter-process exceptions may be signaled synchronously or asynchronously.

A procedure-call-oriented mechanism provides for intra-process exception handling. Error messages provide for synchronous inter-process exceptions. Emergency messages are the key to asynchronous inter-process exception handling.

Streamed connections allow the originator of data to transmit the data to the receiver without waiting for either input requests or output acknowledgments. If the sending process can produce data faster than it can be consumed by the receiver, system-defined flow-control mechanisms are employed to prevent the creation of large message queues [184]. A typical example of streaming in RIG is copying files from one machine to another.

Streaming can be used in any situation in which a synchronous response to input and output requests is not necessary. The advantages of streaming are its low message overhead and higher bandwidth. The major disadvantages are that errors must be signaled asynchronously to the flow of data, and state at both ends must be synchronized in the event of transmission failures.

Four standard procedures are conventionally used for manipulating connections -- Open, Close, Read, and Write. These procedures define the four basic operations on meta-resources or virtual devices. Open and Close deal explicitly with connections (they open and close them). Read and Write may be used for atomic transactions as well. A complete set of message primitives is given in Appendix A.

6.2.3 Asynchronous Messages

Because some catastrophic error situations (such as the death of a process or remote machine) or other exceptions (such as a request from the user to block output to a Virtual Terminal) can occur at any time during a communication, mechanisms must be provided to deal with asynchronous events. Such mechanisms are particularly important for use in conjunction with streamed connections, where the originator of the data does not wait for synchronous responses.

Experience with communication line protocols suggests a polling strategy using timeouts on message primitives as a way to prevent deadlocks in such situations. However, in some cases it is infeasible to place any upper limit on the duration of a transaction, so any timeout-based action will sometimes be invoked when no error condition has occurred. Moreover, in order to avoid the expense of continually checking on the state of a transaction, timeout periods must be relatively large, which makes the prompt detection of errors difficult.

In traditional communications protocols asynchronous events are signaled on a separate interrupt or "out-of-band" communications channel that parallels the data channel [18, 19, 49, 192]. Only very small messages may be transmitted on this channel, typically one byte. Moreover, it is necessary to synchronize the flow of data with the interrupt. For example, if an "abort output" interrupt is sent, a data mark must be inserted into the data channel to delineate data to be discarded.

6.2.1 Atomic Transactions

For atomic transactions, the link between the communicating processes is set up and expires on a message-to-message basis. Process PA simply composes and sends a message to process PB, without PB having to know anything about PA. Depending on the particular request, PA may or may not wait for an acknowledgment from PB. PB retains no information about PA between transactions. Examples of atomic transactions are a request for the time of day, a request for name service, or a request to delete a file.

Atomic transactions have also been used as the base for several distributed file systems and data base management systems [122, 123, 166, 208].

6.2.2 Connections

Any prolonged interaction between two processes may make it necessary for each process to remember the current state of the interaction. In such cases, the processes can create a connection (or session [77, 236]). The primary function of a connection is to control access to a open communication channel. For example, any two processes may read a file simultaneously, but the file system must prevent one process from changing the file position at which the other process is reading. Typically, a server allocates a separate port to each open connection, and associates a state table (connection record) with that port. An incoming request is validated against the table associated with the port on which the message was received, the simplest check being that the sender is the same process that opened the connection.

The amount of state associated with connections varies with the processes involved. Moreover, each end of a connection need not maintain the same state. Where a server typically maintains a good deal of state (the file system, for example, maintains the file position, read/write mode, and other attributes for each open file), the customer process usually remembers only the process and port to which it must send requests, and the port on which it receives replies.

Connections can support two data-flow modes: 1) full hand-shake, or request-driven; and 2) streamed, or data-driven. Full hand-shake means that every data message must be preceded by a request for input and every output message is acknowledged by an "output done" message. Full hand-shake connections are typically manipulated with remote procedure calls. For example, application programs usually maintain full hand-shake connections with their Virtual Terminals so that terminal input and output will be synchronized with respect to program requirements. Full hand-shake has the advantages that the cooperating processes are always synchronized and that the initiator of the connection has complete control of the data flow. However, full hand-shakes are expensive and can easily lead to deadlock (see Sections 6.1 and 6.5).

interface routines and the server itself need be changed. Thus, interface routines further encourage the separation of the semantics of the service from its implementation. Moreover, they enable the service to be documented in the same manner as built-in functions, that is, the name of the request (command), the parameters passed, and the parameters returned.

Typically, a request for service is handled as a remote procedure call [232]. The requesting process initiates a request upon the server and waits for a reply from the server that indicates that the request has been processed (successfully or not) and returns any results. A remote procedure call allows one process to invoke an arbitrary, named (by message id) function in a remote process. Remote procedure calls encourage and facilitate the work of programmers by gracefully extending the local program environment to embrace other processes and machines, thus reducing the cost of adapting existing resources for network use and encouraging the construction of new resources built expressly with remote access in mind.

However, remote procedure calls take away much of the computing power of a distributed system. They impose a strictly sequential execution environment on an inherently parallel system. If used in situations where server replies are unnecessary, they incur an unneeded expense. Moreover, remote procedure calls can easily lead to deadlock (see Section 6.5).

A straightforward alternative is to divide each server call into a request for service (ROS, or simply request) and an optional completion of service (COS, or simply reply). Separate procedures can be associated with each phase. The request is formatted and dispatched by an ROS procedure, whereupon the sending process can continue. If a response is received, the appropriate COS routine is called (synchronously or asynchronously). Such a scheme is typical of multiplexed processes, and is often more difficult for programmers to master (see Section 6.4).

6.2 Communication Styles

When two processes wish to communicate they are free to do so in any mutually convenient manner. Experience with RIG indicates that three fundamental styles of message communication may be sufficient:

1. atomic transactions
2. connections
3. asynchronous messages

CHAPTER 6

Multi-process Structuring

Previous chapters have concentrated on the architecture for RIGITS. Fairly precise definitions have been given for functions that several of its components -- VTMS, the command interface, the Process Manager -- must perform. This chapter discusses methods whereby those functions are implemented.

In RIGITS modular decomposition via processes and messages has eased the writing of new processes, but forced them to obey a substantially different set of rules than in a typical sequential program execution environment. Various styles of intercommunication must be supported. Because of the "open" environment, each process is fundamentally responsible for protecting itself. Providing convenient mechanisms for both intra- and inter-process exception handling relieves the individual programmer of providing his own. In sum, principles of multi-process structuring [39] are crucial to the success of a distributed system.

6.1 Server Calls

A server provides access to a type of resource, such as the file system. If the particulars of the resource change, such as the disk drives supported, the server must be modified. However, the interface presented to the rest of the world should not change unless the fundamental characteristics of the resource change (random access is now provided). Therefore, it is generally unnecessary to alter any code outside the server when the server itself is modified.

Customer processes may also be isolated from syntactic changes in the formats of messages sent to servers. This buffer is provided by the use of a run-time linkable interface package for the server. In general, each server process will provide at least one interface routine that accepts a request with data, formats an appropriate message and sends it to the server, and returns results from the server.

If processes access a server only through interface routines, rather than generating their own messages, it is much simpler to change the protocol for that service. Whenever protocol changes, only the

build a robust distributed system, and various levels of protection have been added (see Chapter 6). Capabilities, in particular, are being employed in various descendants of RIGITS [32, 174].

The primary contribution of RIGITS is its mechanisms for process management. Creation of processes by name and registration facilities were relatively simple embellishments that are infrequently found in other systems (Thoth, DEMOS, and StarOS provide similar facilities). The registration facilities, in particular, have been invaluable for managing distributed jobs and are discussed further in Chapter 6. The RIGITS Job Managers collectively act to form a single resource manager or executive for the whole system, as proposed by Jensen [108].

in Section 4.4.2. The execution of a job step is managed by the Job Manager associated with the particular RIGITS machine. When a job involves resources distributed across multiple machines, the Job Managers communicate with each other to manage the collective resources and to determine the best distribution.

The Job Manager performs the following fundamental tasks:

1. maintenance of the state of all jobs associated with the local machine, i.e., all services allocated to each job;

2. provision for recovering all resources when a job terminates;

3. authentication of requests for resources, based on the access rights contained in the User Profiles;

4. resource selection on the basis of user-supplied requirements and system-supplied information such as load;

5. maintenance of all user-oriented system state;

6. collection of statistics.

Recovering resources is simplified by the process management strategies discussed in Section 5.1. All processes associated with a particular job may be killed by requesting the Process Manager to terminate the process sub-tree associated with the job. Any other processes (typically servers) that were communicating with the job's processes will be notified that the job has terminated (i.e., certain processes have died), and can then release allocated resources.

Authentication and resource selection have received little attention in RIGITS. Much of the relevant information is provided by User Profiles (see Section 4.4.1). The User Profile contains the systems the user is permitted to access, and the types of interaction allowed. The User Profile is accessed by the Job Manager when the associated user logs in to RIGITS, and may be permanently changed only by the Job Manager at the request of the user. Some additional remarks may be found in [72].

5.3 Historical Perspective

In some sense, resource management has received relatively little attention because the basic function of mutual exclusion is provided so well by the intrinsic modularization. Also, RIGITS was originally designed for an environment where programmers and users alike were assumed to be kind, courteous and competent. Little effort was expended on protection and the resulting system was considerably simplified. However, chivalry has not proven a very firm foundation upon which to

5.1.3 Process Registration

Any process, PA, may request that it be notified when particular events occur with respect to another process, PB. In terms of process management, these events include process suicide, crash, suspension, and resumption. (If the Process Manager is forced to kill a process, this is considered to be a process crash.) Presumably, a process will wish to be notified about a death in the family. Process PA may also request to be notified if a process PB not in the immediate family dies (or is suspended), or to notify PB if PA crashes — if, for instance, PA is an application process with open files and PB is the file system.

The Process Manager relies on the Kernel to notify it whenever a process dies, is suspended, or resumed. The Process Manager then distributes the appropriate emergency message to all interested parties. Notification by emergency messages ensures that a process, PA, will not sleep waiting for a process, PB, that may have died.

Further details of registration and emergency messages may be found in Chapter 6.

5.1.4 Access Control

In order for a process, PA, to communicate with another process, PB, PA must obtain PB's address. PA may obtain the address by broadcasting a request for service to which PB responds (generally via a name-server — see Section 2.2.3.3). Alternatively, a process, PC, may send to PA the address of PB in a message. This latter method presents a problem of access control: PB may not want to communicate to any process that does not specifically request its services.

To provide access control, process addresses must be protected. Using capabilities, PA may only communicate with PB if PA has been granted an explicit capability to do so. Capabilities are being incorporated into the interprocess communication facilities of UNIX [174], an effort that is a direct descendant of RIG. Capability-like functions are currently provided by connections (see Section 6.2.2).

Additional access control facilities are provided in terms of user jobs.

5.2 Job Control

Satisfying a user request entails performing a job. A job represents an encapsulation of all the resources associated with a particular user activity. These resources may be distributed throughout the network. It is necessary, first, to provide a job control language that allows the user to specify a collection of job steps, each of which may be executed on a different machine, and second, a means of executing the constituent job steps. Job control languages were briefly discussed

call. The Process Profile is used to fill in any parameters not specified in the call. The resulting Process Definition Table is passed to the system Kernel, which allocates the virtual address space for the process, reads in its code segments, and starts the process.

It is possible to download a process onto a remote host by specifying a particular host in the Process Profile. When the PDT is passed to the local Kernel, the request will be routed to the Kernel on the remote host (see Section 2.2.3.2).

5.1.2 Process Termination

Processes may die in one of three ways: suicide, accident (crash), and homicide. When the local Kernel notes the death of a process, that information is passed to the Process Manager. Network servers are responsible for notifying the Kernel of the deaths of remote processes.

A process, PA, or ancestor thereof, may request, at any time, that PA, or PA's descendants, or PA and its descendants, kill itself (themselves). Such deaths are termed "clean" or "normal" in that the process is requested to kill itself, thus giving it the chance to clean up its world. If a process, once doomed by such a request, does not kill or detach itself within an allotted time, the Process Manager is responsible for murdering it.

Only the Process Manager has the authority to request a process's death, or to murder it. Processes should protect themselves against termination requests from any other process. No process may be murdered without first giving it the opportunity to kill itself. This policy guarantees that each process has an opportunity to clean up its state, with a greater possibility that resources are deallocated properly. (The policy reflects early experience with RIG. The process registration facilities discussed in the following section and Chapter 6 may eventually eliminate the need for such chivalry.)

When a process dies, it is removed from the process tree and the care of its sons must be given to other processes. In particular, each son is placed in the care of his godfather, or, if the godfather is also dead, the system process. The father's godfather (or the system process if that godfather is dead) becomes the new godfather for the sons.

After death, a process is left in the tree for a period of time, although marked as dead. This allows an Executive or Monitor, for example, to clean up the dead process's subtree, or to ask for the key (name) of the dead process for logging purposes.

5.1 Process Management

The Process Manager is responsible for the creation, registration, and destruction of processes associated with a particular RIGITS machine. There is one Process Manager per machine, and it may be a separate process or a part of the Kernel. All (other) processes are created via a request to the Process Manager. Any process wishing to kill another process must make this request of the Process Manager. Registration enables a process, PA, to specify that it would like to be notified when an exceptional event occurs with regard to another process, PB — e.g. PB dies or is suspended.

The Process Manager maintains an explicit tree of processes wherein each process is regarded as the son of some other process, typically the process that created it or the system process, namely, the Kernel. In contrast to many systems, the process tree imposes no restrictions on the ability of processes to communicate. The tree is used to group all processes created in response to particular jobs. When the job is terminated, all processes in that subtree are terminated.

When a process is created, a "godfather" may be specified that is to assume responsibility for the process should its "father" die. Godfathers are typically grandfathers. The notion of godfathers is intended to provide a "responsibility chaining" mechanism, whereby higher-level processes may institute administrative actions should catastrophic events disrupt the normal process structure.

Sons of the system process fall into two groups: those that have been detached, and those that have been orphaned due to the death of their fathers. Detached and orphan processes are scavenged at regular intervals to prevent them from tying up system resources.

Detached processes provide the ability for suspending user programs for resumption at a later time. A process may request at any time that it be detached. In particular, when asked to kill itself, the process may instead ask to be detached. When the process is detached, its entire subtree is grafted onto the root (system) node. The management of detached processes in a distributed environment has not been extensively studied and remains an important research question.

5.1.1 Process Creation

Associated with each process, PA, is a Process Profile describing the generic process of which PA is an instantiation. In addition to specifying the code segments to be loaded, the Process Profile contains the default process attributes such as priority, stack size, free memory, and number of ports.

A process is created by passing the appropriate process key (such as "Executive") to the Process Manager. The Process Manager uses the key to access the appropriate Process Profile. Any parameters intended to override the default profile parameters may also be specified in the

CHAPTER 5

Resource Management

Provided with the facilities to use any number of tools simultaneously, the user can build up quite a set of activities, or jobs. Each job is allocated a dynamically changing set of resources such as processors, memory, terminals, files, and processes. To protect users from each other and the system from users, and to enhance performance, facilities must be provided for managing these resources. "'Management' refers to the decisions and actions involved in resource utilization -- how they are assigned and released, altered and consumed" [108, p. 1].

Resource management is a central design problem for any operating system. In RIGITS, the basic resources include buffer space for messages and Kernel data structures, and processor cycles. By controlling the allocation of these resources, the RIGITS Kernel controls the rate of message flow between processes and guarantees that each process is granted its "fair" share of processor time. The necessary flow control, reliable transmission, and scheduling mechanisms are beyond the scope of this thesis (see [185]).

Above the Kernel, system resources -- such as disk files, printers, terminals, tape drives, and network links -- are associated with server processes. When considering the integrity of a resource, it is unnecessary to consider any process other than the associated server(s). The problem of resource management at this level is fundamentally one of creating, destroying, and granting access to processes. These functions are primarily the responsibility of the RIGITS Process Manager. In addition, each server process must protect itself against invalid requests and manage its own data structures properly. Conventions and paradigms for process structure and intercommunication are discussed in Chapter 6.

Another aspect of the resource management problem is the ability to provide services to user jobs and to recover those resources when the job is terminated. These functions are provided in large part by the RIGITS Job Manager.

4.6 Historical Perspective

This chapter presents a design for a command interface originally proposed in May 1977. Although largely a matter of personal preference, it was greatly influenced by systems such as TENEX and TOPS-20, the work on NLS and the Command Meta Language at SRI International, and PLITS. Coincidentally, the command interpretation language is almost identical to the System/38 Control Language [26], which was developed at roughly the same time.

Man-computer communication has been a subject of major research at several other institutions. The Language for Conversational Computing (LCC) is an example of an early effort at defining a sophisticated command language [153]. PACE is an example of current efforts [3]. USC/Information Sciences Institute has developed formal methods for "protocol" analysis of command languages [88, 89], and the concept of "dialogue games" [143]. LIFER is an outstanding entry in the area of natural language interfaces [90, 91].

The National Bureau of Standards has developed one of the most sophisticated command languages extant for its Network Access Machine [188], and has been seriously involved in the problems of user interaction for well over a decade. Rand's RITA and Exemplary Programming Projects are designed to provide programmable network access services that "learn" [4, 5, 223, 224, 225]. Both NAM and RITA make use of unmodified constituent hosts and use string substitution to translate a user's generalized command into host-specific commands. Both provide a capability for analyzing the response to a command so that conditional statements may be executed.

The primary contributions of the RIGITS command interface are its command interpretation language and its (suggested) implementation. The command interpretation language is amenable to various grades of user proficiency and avoids the complexity and ambiguity of current natural language interfaces. The implementation guarantees fast, event-driven response to user interaction, and simplifies the task of writing user interfaces for pre-existing tools. The separation of function between the Tool Interface Process and the Tool Service Process(es) corresponds to the distinction between Frontend and Backend processes in the National Software Works. The combination of a Tool Interface and a Command Profile corresponds to a Command Meta Language (CML) grammar (which results from the compilation of a CML program) [106]. Whereas CML provides limited error detection and recovery capabilities, the RIGITS design provides the user every opportunity to correct his ways when errors are encountered. However, having a meta-language with which to define such interaction is an attractive feature.

4.5 Implementation Caveats

The suggested implementation of the command interface requires that a large number of processes be created and destroyed in the course of interacting with the user. Depending on the supporting operating system, process creation and destruction may be expensive. If this is the case (as indeed it is in RIG), each Tool Interface or Tool Service Process need not be implemented as a separate process. Moreover, for pre-existing tools, the TSPs may not be implemented as processes, yet RIGITS must still be able to communicate with them.

It thus becomes more appropriate to refer to Tool Interface and Tool Service Modules (TSM) as opposed to processes. TIFs in particular may then be represented as subroutines. The resulting flow of control (Figure 11) is quite different from that shown in Figure 10. TIF1, for example, communicates with CI1 to determine which command to invoke. It may then call TIF2 as a subroutine. TIF2 may require the same CI1 to complete or correct the command. Only if TIF2 provides subcommand would it require another Command Interpreter CI2. CI2 could be separate process, or CI1 could be multiplexed to act as CI2 (see Section 6.4.1). In any case, when TIF2 returns control, TIF1 initiates another request to CI1.

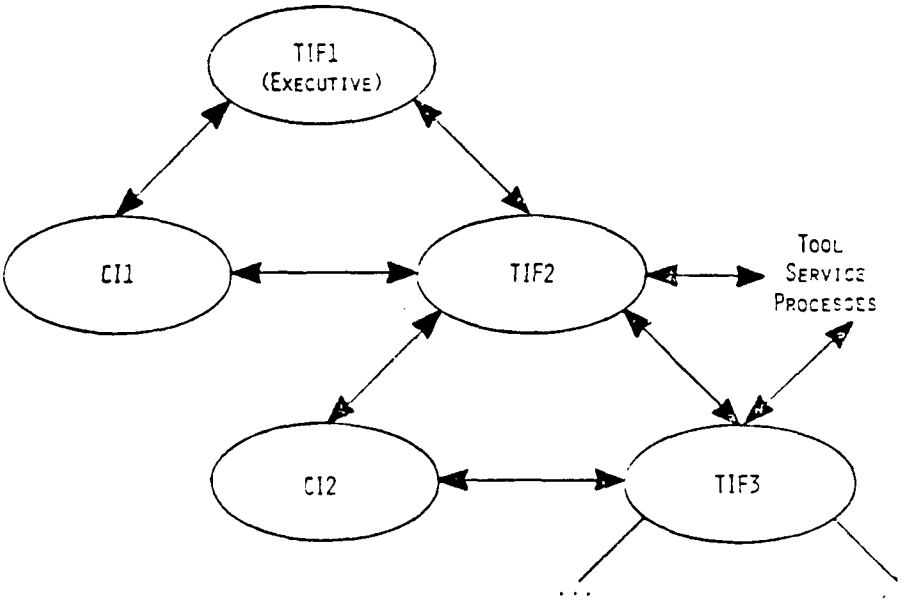


Figure 11. An alternative command interface.

An orthogonal consideration regards Virtual Terminals. A tool may employ different Virtual Terminals for different types of interaction. It may therefore be appropriate to associate a Command Interpreter with each Virtual Terminal, such that the TIF need only refer to a single entity, the Virtual Terminal, when interacting with the user.

accessible tool the user may specify his own Command Profile. The template for a User Profile is given in Appendix D.

4.4.2 User Programming

User programming allows the user to avoid tedious repetition of commands. The available facilities should include programmable function keys, macro files, and command programs.

4.4.2.1 Programmable Function Keys -

Programmable function keys are straightforward as long as symbolic arguments are not allowed. The keyboard driver (microcomputer) simply generates a continuous, uninterruptable stream of characters to the VTC, just as if the user had typed them. If arguments are introduced, it becomes necessary for the VTC to parse the programmed string and replace symbolic arguments with actual keyboard input. In any case, programmable function keys are processed whenever they are typed, and can not be aborted.

4.4.2.2 Macro Files -

Macro files provide the capability of substituting a file for keyboard input. Two types of macro files may be of use. The first type simulates a programmable function key by taking effective control of the keyboard; this is useful for executing "transcript" files of previous sessions or initiating a particular series of tools upon logging in. The only keyboard function that can be fielded while the indirect file is being processed is ABORTTASK.

The second type of macro file replaces keyboard input only for the active Virtual Terminal. In addition to ABORTTASK, all keyboard functions necessary for managing Virtual Terminals will continue to be fielded.

In either case, the same difficulties with symbolic arguments carry over from programmable function keys.

4.4.2.3 Command Programs -

Command programs require a command programming language and command language interpreter. Command languages are a topic of ongoing research at Rochester [141].

6.3.1 ErrorSet and Error

Within a single process a procedure-call-oriented mechanism (ErrorSet and Error) allows an error notification to propagate up the "calls" hierarchy [165] to specified points. ErrorSet is a function that accepts a severity level and a procedure as arguments. Error accepts a severity and error code as arguments. ErrorSet calls the procedure and traps all errors of the specified severity or lower: If Error is called, the call stack is unwound to the point of the most recent ErrorSet with a severity level equal to or higher than the severity level of the Error. The error code is returned to the caller of ErrorSet, which can then attempt to recover from the error.

Since a call to Error can occur deep within a set of nested procedure calls, it may cause many procedure activations to be removed from the stack. If the Error call is not trapped the process is terminated. There is no notion of explicit responsibility chaining at this level. If, however, the process is terminated, interested parties can be notified via the process registration facilities (see Sections 5.1 and 6.3.4).

6.3.2 Error Messages

Many types of errors are generated synchronously in response to a request. These errors are frequently reported back to the requesting process via an error message -- indicated by an ERRMSG message id. When an error message is received, it is typically used to generate another Error call, thus converting back to the intra-process exception mechanism.

6.3.3 Emergency Messages and Handlers

Asynchronous inter-process exception handling is supported by the RIG emergency message. Because emergency messages are delivered asynchronously, and will "awaken" a blocked process, they provide for event-driven, prompt error recovery. Emergency messages are fielded by emergency handlers.

In RIG, only one emergency handler may be associated with a process at a time. When an emergency message arrives for a process, the emergency handler will be called with the message as its argument. (Note that the handler is only invoked when the process is attempting to send or receive a message.) The handler may process the emergency in any manner it sees fit, including performing an Error call to some higher level in the process's control path.

Typically, an emergency handler returns a Boolean value indicating whether or not to abort the request in progress at the time the emergency arrived. For example, if the process, PA, is awaiting a reply from a process, PB, when an emergency message arrives telling of PB's

death, the request should be aborted. A typical emergency handler is shown in Appendix E.

This relatively simple approach to emergency handling is attractive for several reasons. It is exceedingly cheap to implement. Because only one emergency handler is active at any time and the process implementer must explicitly change it within the code, the implementer rarely loses track of which handler will be invoked. Finally, the context of a particular invocation is perfectly understood: The process is performing a message send or receive primitive and has inadvertently received an unexpected emergency. If the emergency is handled without error, the send or receive is continued; otherwise the intra-process Error facility may be employed as if the send or receive had failed.

6.3.4 Event Handlers and Spooling

Emergency messages may be generated by any process, but are typically generated by event handlers. An event handler is a process that is capable of detecting, or that will always be informed about, the occurrence of a particular kind of event. In general, a process, PA, must register with an appropriate event handler that it wishes to be notified when a particular event, EPB, occurs with respect to process PB. When EPB occurs, the event handler notifies PA via an emergency messages.

One particular event with which all processes are concerned is the death of other processes and machines with which they are communicating. The RIGITS Process Manager provides for notification in the event of process death, suspension, or resumption (see Section 5.1.3)

Typically, a process might want to take a specific action when a particular set of resources becomes available. The availability of each resource may be termed a "simple" event. Sophisticated event handlers should allow the specification of Boolean combinations of simple events. These handlers might be termed spoolers since they buffer (spool) the constituent events until the entire combination is completed. Spoolers would themselves employ simple event handlers for each "simple" event. The identical approach has been suggested by Cheriton [39], although he relies on processes blocking until the desired events occur.

6.4 Process Structure

The mechanisms presented above facilitate process intercommunication. Because the internal structure of a particular process is transparent to any other process it has not been necessary to discuss details of process structure. This section presents some methods that have proven useful in building robust processes. A prototypical process is shown in Appendix F.

6.4.1 Ephemeral vs. Permanent

Each RIGITS server process is designed to perform a particular service. Either a new instantiation of the server must be created each time the service is requested, or one server can be programmed to provide the service simultaneously to multiple requesting processes. The two variations may be termed ephemeral and permanent. An ephemeral process is typically created by the name server in response to a request for service (see Section 2.2.3.3), or by a "higher-level" server that wishes merely to dispatch a sub-task to another process.

A permanent server offers several advantages. In general, it uses significantly less memory and requires less scheduling overhead than the corresponding group of ephemeral servers. In some cases (such as the file system) centralized control is required to synchronize and arbitrate interactions among tasks that must share resources; here, a permanent server solves the mutual exclusion problem (there is, after all, only one disk channel).

However, a permanent server must ensure that it is never deadlocked, and rarely suspended waiting for a sub-task to complete. This suggests that a permanent server is typically multiplexed.

6.4.2 Dedicated vs. Multiplexed

In a multiplexed server, a single transaction may be partially processed, then halted to wait for a sub-task to complete (or simply to ensure "fair" access to the resource). This requires saving the intermediate state of the current transaction so that work can continue on other requests. Thus, a multiplexed server normally maintains, for each outstanding connection, a data structure that characterizes the current state of the connection. (This state is often associated with a particular port of the server process, and that port is used for all communications with the user process (see Section 6.2.2).) Multiplexed servers are therefore larger and more expensive to swap in.

Dedicated servers, on the other hand, process each request to completion. A request for execution of a sub-task is typically executed as a remote procedure call. Dedicated servers are usually much easier to write than multiplexed servers.

6.4.3 Crash Protection

Error and emergency messages and the Error function are the mechanisms whereby errors may be posted. Emergency handlers and ErrorSet are the corresponding means whereby those errors may be fielded. Every process must specify an emergency handler to cope with errors from external sources.

In addition, every process should protect itself against all errors generated in response to fulfilling a request from another process. Insofar as possible, they should validate the requests made of them. This is made much simpler by the fact that variables and procedures are not shared between processes.

One approach is to code processes in hierarchical levels -- each level performs an ErrorSet call on the level below it, usually with different severities. In RIG, at least three levels are employed: The third (lowest) level performs all message processing; the second level initializes state, and receives and dispatches messages; the first level serves merely to trap errors at all lower levels. If an error is caused by an invalid request, the third level may perform an Error call which will be fielded by the second level; the request will then be flushed and the requesting process notified of his error. Memory and stack errors, however, may evade this mechanism (due to their higher severity), and pop to the first level, where cleanup is instituted and the process aborted.

6.5 On Deadlock

Deadlock remains a perplexing problem in distributed systems. In RIGITS the only foolproof way of preventing deadlock is to disallow processes from blocking when they attempt to send messages, or to impose mandatory timeouts. Although this approach guarantees that processes will never go to sleep (or stay asleep) waiting for each other, they might spend all their time in polling loops.

However, the Kernel can prevent certain deadlocks. The most obvious deadlock is when each of two communicating processes is executing a remote procedure call on the other. The Kernel can detect this circumstance and prohibit the second party from initiating a request on the first. However, if one of the processes is remote, the Kernel must communicate with a network server, which can prove expensive.

Another type of deadlock results from flooding of message queues. If process PA attempts to send a message to a full port (queue) of PB, PA will be blocked, whether or not PA is attempting a remote procedure call. If PB then attempts to send a message to a full port of PA, PB will be blocked. Mechanisms for setting the queue capacities reduce the likelihood of this type of deadlock, but do not eliminate it.

Deadlocks involving more than two processes present additional complexities. The interested reader is referred to [37, 39, 98].

6.6 Historical Perspective

In the best of all worlds the programmer of a distributed system would be provided with a language for distributed computing. The primitives of such a language would include processes, messages, clocks, transactions and the like. PLITS [71], ZENO [14], *MOD [45], LIMP [103], and TASK [113] are representative of current efforts in this direction. Note that effective programming languages for distributed computing should have much in common with the network job control languages mentioned in Chapter 5. See Mohan [156] for a survey.

Such languages were not available when we began to implement RIG. It was necessary to build up from an existing base language to provide the distributed programming environment. The mechanisms we have outlined above reflect in part the environment imposed by BCPL [181].

The mechanisms presented here fit under the rubric "mid-level" protocols. They would be implemented, for example, in levels 5 and 6 of the ISO Standard for Open Systems Architecture [58, 236], or in the Service Support Layer of LLLNOS [77]. The Distributed Processing System employs remote procedure calls to extend the local programming environment to remote machines [231, 232]. Multi-processing structuring in general is the subject of numerous papers, including [11, 39, 102, 164, 183, 219]. RIG reinforces many of the ideas presented in these papers with a great deal of practical experience.

Levin has developed one of the few formal models of exception handling [133]. For intra-process exception handling, Levin's mechanisms would prove more flexible than the RIGITS ErrorSet-Error, or similar mechanisms such as those provided in INTERLISP [211]. Because they employ shared memory and contexts, his mechanisms are not easily extended to a distributed environment. Extended-CLU is a step in this direction [136, 137].

Since their inception, communications protocols have employed asynchronous interrupt signals, but typically require that interrupts be synchronized with the parallel data channel. Similarly, MSG provides "alarms" which arrive on channels distinct from the normal message medium [162]. Streamed connections are quite common in network protocols — such as MSG connections, the Ethernet Byte Stream Protocol, and TCP [169].

The registration facilities provided by RIGITS, and exemplified by their use in process management, are unique to RIGITS. Various systems, including DEMOS, provide the option of being notified when a "son" dies. Cheriton proposes mechanisms similar to event handlers for Thoth.

CHAPTER 7

A Case Study: RIG

The preceding chapters have presented a design for a sophisticated message-based distributed system, RIGITS. This chapter concentrates on the current implementation, RIG, which represents roughly eighty percent of the RIGITS design. I will first outline the major discrepancies between RIG and RIGITS, followed by an extended example that will demonstrate most of the concepts developed in the body of the thesis.

7.1 RIG vs. RIGITS

The RIGITS distributed operating system is supported primarily by the gateway Eclipses. All other machines are loosely integrated in that they currently run some version of a Kernel and represent various degrees of RIGITS hosts. RIG is currently a user host, not a server, on the ARPANET. As work progresses, the Altos, VAX, and DEC-10 will become better integrated. In particular, the network access program currently being used on the Altos (Nexus [74]) is being replaced by a new multi-processing operating system [32], and a new interprocess communication facility is being incorporated into VAX/UNIX [174].

7.1.1 System Architecture

7.1.1.1 System Super-structure -

Each gateway Eclipse supports the following fundamental components:

- Kernel
- Process Manager
- Job Manager
- File System
- Name Server
- Ethernet Server
- Logger process
- Timer process
- Console Server for the system console

- on a per user basis:
  - a Virtual Terminal Controller
  - a Monitor
  - Executives
  - tools

Together, the gateway machines provide the following services:

- DEC-10 servers
- ARPANET servers
- Spooler
- Versatec server
- Tape server
- GMR (color display processor) server

The location of these services is invisible to users and processes alike.

7.1.1.2 Resource Management -

Jobs are not currently managed in any coherent fashion. They are implicitly associated with subtrees of processes maintained by the Process Manager. The Job Manager's primary function is to activate terminals associated with its host upon user request.

7.1.2 The User Interface

The user currently views distributed RIG in a manner similar to RSEXEC. The file system is distributed in the sense that hosts have been added to the naming hierarchy. "Connected" directories allow the user to specify partial pathnames. Multiple copies of files are not maintained, nor is there a master directory saying where the files reside.

There has been no attempt to provide a distributed, multi-computer access control and accounting system, nor is there any automatic reconnection (login) mechanism. Many of these issues impact on the definition and use of User Profiles.

7.1.2.1 Virtual Terminal Management -

Several editing features are not supported by the Pad: cursor motion by words; deletion of words or pages; joining and splitting lines; text selection and transfer; insertion of files at any point. (The last three sets of functions are supported by the RIG screen edi-

tor.) By default, Virtual Terminals do not save their output on scratch files because the RIG file system currently can support only a limited number of open files. Therefore, viewing cursors and scrolling are not supported, nor can output be suspended or discarded, directly by the Pad.

7.1.2.2 The Command Interface -

The RIG command interface is provided by a set of subroutines rather than independent processes. As a result, the current command language is strictly positional and token-oriented. Completion and recognition, type-checking, and defaults are provided. An attempt is made to provide on-line assistance (lists of options or syntax) at any point during command input. Indirect input (programmable function keys, macro files, command programs) is not supported. Context is not maintained.

Because there are no Command Interpreters, Command Profiles are not used. TIFs are typically implemented as subroutines of the Executive.

7.1.3 Profiles

Profiles are currently supported for terminals and processes.

7.2 The User View

The basic logical flow of RIG, to a single user, appears as follows:

1. When the terminal is inactive, it displays an appropriate message. In response to any character it is "awakened".

2. A Monitor and Status Server are created for the user. The base image appears, and the Status Server displays the system news file.

3. The Monitor processes commands for managing Executives and allocating screen space. If the "Quit" command is issued, all resources associated with this terminal are released and control returns to (1). Assuming an Executive is given control -- via the "SpawnExecutive" or "ResumeExecutive" commands, or the various screen management keys -- the screen is remapped to contain an instance of the Executive and...

4. The Executive processes commands for file management, and the like. If the "Quit" command is issued, the Executive disappears, and the screen may be remapped to activate another

Executive or the Monitor. If a request is made, for example, to edit, or talk to the DEC-10, a tool is forked. The tool will run in the same window allocated to the Executive.

5. The tool executes, interacting with the user as necessary. Upon satisfactory completion, the tool dies, whereupon control is returned to (4).

6. At any time the user may direct his attention to a different tool, Executive, or Monitor.

7.3 The System View

The internal flow corresponding to a typical user session is as follows:

1. The Job Manager listens to all "inactive" terminals. Each such terminal has a Terminal Input and Output Handler associated with it. When any character is typed, the terminal is "awakened." It is also possible for a remote machine (e.g., an Alto) to send a REQUESTFORLINEMSG. For each active terminal or remote line, steps (2)-(6) are executed.

2. A Monitor is created for the user. The Monitor initializes the Virtual Terminal Controller for the user, and spawns a Status Server.

3. The Monitor processes commands for managing Executives, and allocating screen space. For each Executive created, steps (4)-(6) are executed. When the Monitor is killed, its entire subtree of processes is killed as well. Any resources associated with the user are deallocated, the terminal becomes inactive, that is, control for that terminal returns to (1).

4. The Executive interacts with the user to specify a command. It then invokes the appropriate Tool Interface Module (TIF), either as a subroutine or a process.

5. The TIF interacts with the user as necessary. When necessary, the TIF may spawn one or more Tool Service Processes (TSP) or initiate communication with already existing TSPs. For file management commands, the local file system and remote file servers serve as TSPs. TSPs rely on the TIF to interact with the user in order to correct errors or collect additional input.

6. When the TIF finishes, typically by user request, any processes created by it are killed and control returns to (4).

7.4 An Example

When the user first accesses RIG, a Monitor and Status Server are created for him and mapped to the screen as shown in Figure 12 (all figures are Versatec plots of a Delta 4000 terminal screen). The Status Server is mapped to the upper 5-line Region composed of a 1-line Viewport for the RIG banner, and a 4-line Viewport for RIG status information. The Monitor is mapped to the lower 20-line Region composed of a 1-line Viewport for the Monitor banner and a 19-line Viewport for Monitor command interaction. Hence, the Status Server and Monitor each possess two Virtual Terminals.

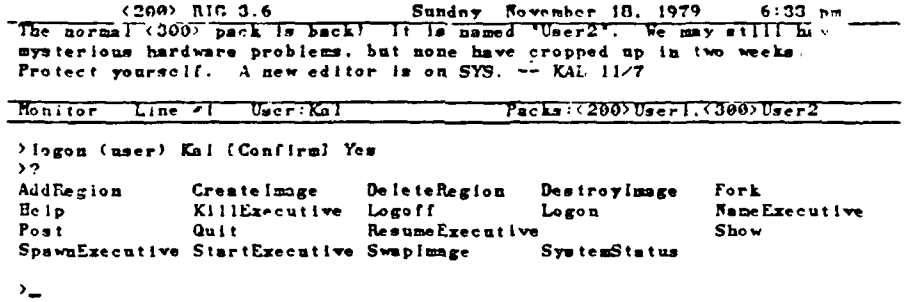


Figure 12. The base Image

At this point, exactly six additional processes have been created for the user. The Monitor is the root of the tree of processes. For each Virtual Terminal, the Monitor or Status Server maintains a logical connection to the appropriate Virtual Terminal Controller. The logical connection subsumes multiple connections with the appropriate Line, Pad, and Screen Handlers. Similarly, the processes comprising the Virtual Terminal Controller maintain connections among themselves.

The available Monitor commands deal with Images, Regions, and Executives. Note that the EXPAND and PROMPT keys discussed in Section 4.1.5 have been implemented. Because command interaction is currently implemented as a set of subroutines, the CANCEL key results in an intra-process Error call. ABORTTASK, on the other hand, results in an emergency message being delivered from the Virtual Terminal Controller. The typical way of starting an Executive is "SpawnExecutive;" this creates the Executive, creates an Image for it, and swaps to that Image.

This Image consists of one Region to which the Executive is mapped (Figure 13). The Executive, like the Monitor, possesses two Virtual Terminals. The banner Virtual Terminal is mapped to the 1-line Viewport; the command interaction Virtual Terminal is mapped to the 24-line Viewport. The available Executive commands manage files, and run various tools, e.g., edit, and TELNET. Wherever a command verb is specified, the Executive sends an emergency message to the Monitor containing an indication of the command to be executed. Thus, the Monitor keeps track of what activities the user is engaged in.

```
Exec02/edit/                                Directory:<200>kal

>?
ArpaTelnet      ChangeAttributes            Compare         Compile
Connect         Copy            CreateDirectory             DecTelnet
Delete          Directory       Display     Edit            Fork
Help            Load            Post        Print           Quit
RemoveDirectory                 Rename      Run             Set
Type

>connect (with directory) kal
>edit_
```

Figure 13. An Executive

At this point, the Executive is added to the process tree as a son of the Monitor. An additional Pad Handler is created to handle the output requirements of the Executive's Virtual Terminals.

When the editor is run, it "replaces" the Executive's command interaction Virtual Terminal with three of its own, one for status, one for command interaction, and one for text-editing (Figure 14). Together with the old banner, these four Virtual Terminals comprise a new Configuration of the Superwindow associated with the Executive. The user changes between the command and text Virtual Terminals via the CHANGEVIEWPORT key.

At this point, the editor TIF is added to the process tree as a son of the Executive. An additional Pad Handler is created to handle the output requirements of the text-editing Virtual Terminal. No additional Tool Service Processes are created. When a file is being edited, the editor maintains a connection with the file system corresponding to the open file. At the time the editor is created, the Executive registers with the Process Manager that it wishes to be notified if the editor dies.

```
Exec02/edit/Editor                    Directory:<200>ka1
Locate:                                        File: <200>ka1:novarc.sa1
Change:                                        Mode: Insert

Command?
require "ALL:CPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 NewItems;
require 2000 PNames;


   ! Tuning parameters;

define  INPUT!BUFFER!SIZE       = 256,
        HEADER!FILL!CHARACTER   = '55,
        PAGE!LENGTH             = 58,
        PAGE!WIDTH              = 132,
        COMMAND!PROMPT          = [ CRLF & "Command? "];

   ! Dump file info;
```

Figure 14. The RIG screen editor

Assume the user has finished editing a program that will run on the DEC-10. He wants to transfer it to the DEC-10 and compile it, but can fix any errors in the local file maintained on RIG. To do so, he can map DEC-TELNET and the editor to the same Image. He first returns to the Monitor via the RESUMEMONITOR key. Using the "StartExecutive" command, he starts a new Executive from which to run DEC-TELNET. "CreateImage" creates a 10-line Image containing it. The editor is added to the Image with "AddRegion" -- a 15-line Region for the editor is added to the 10-line DEC-TELNET Region to create a 25-line Image. See Figure 15.

This Image may be activated with "SwapImage" or the CHANGEIMAGE key (Figure 16). The user may change between DEC-TELNET and the editor with the CHANGEREGION key. The user runs DEC-TELNET and transfers the source file to the DEC-10. When compiling, an error is found. The necessary change is made in the local file.

At this point, the DEC-TELNET TIF has been added to the process tree as a son of the Executive. Figure 17 shows the tree of processes associated with the user. The link to the DEC-10 may be managed by either of the gateway Eclipses; to find the appropriate server process, the DEC-TELNET TIF broadcasts a request for service to the name-server and opens a connection to the indicated process. The DEC-TELNET server registers with the Process Manager that it wishes to be notified if the TIF dies.

After the compilation is finished, the user closes the file and transfers it to the DEC-10 again. There it is recompiled without errors.

```
        <200> RIC 3.6            Sunday  November 18, 1979     6:48 pm
The normal <300> pack is back!  It is named "User2".  We may still have
mysterious hardware problems, but none have cropped up in two weeks.
Protect yourself.  A new editor is on SYS. — KAL 11/7

Monitor   Line #1   User:Kal                Packs:<200>User1,<300>User2
SpawnExecutive StartExecutive SwapImage      SystemStatus

>spawnExecutive (name) edit...working
     => Exec02
>s?
Show            SpawnExecutive StartExecutive SwapImage      SystemStatus

>StartExecutive (name) compile...working
     => Exec03
>createImage (executive) Exec03 (lines) 10
     => Image 2
>addRegion (executive) e?
Exec02          Exec03          edit            Editor225
 (executive) Edit (image) 15 (lines) 2 (region index)  XXX

>addRegion (executive) edit (image) 15 (lines)  XXX

>addRegion (executive) edit (image) 2 (lines) 15 (region index) 2
>_
```

Figure 15. Creating an Image.

```
Exec03/compile/TenTelnet                Directory:<200>SYS
  TTYUTS.HDR[170,166] 1
  SCNUTS.HDR[170,166] 1
  GPUTS.HDR[170,166] 1
  DATES.HDR 1
INSERTING FORGOTTEN SEMI-COLON.
TEST.  PAGE 1
04000       require 2000 NewItems;

^
Exec02/edit/Editor                      Directory:<200>kal
Locate:                                         File: <200>kal:novarc.sai
Change:                                         Mode: Insert

Command?
require "ALL:SCNUTS.HDR[170,166]" source!file;
require "ALL:GPUTS.HDR[170,166]" source!file;
require "CS:DATES.HDR" source!file;

! require "SUB360[170,166]" load!module;
! external proc List360 (string filename);

require 2000 New!Items;
require 2000 PNames;
```

Figure 16. Editing and compiling simultaneously.

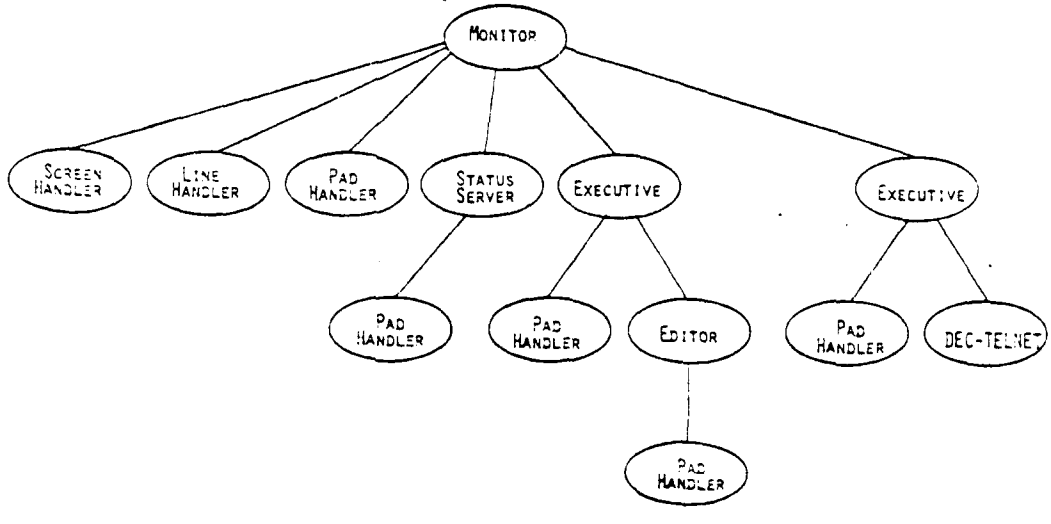


Figure 17. The user's tree of processes.

At this point, the user wants to go home. He returns to the Monitor via the RESUMEMONITOR key and displays his current status (Figure 18). Having decided that there is nothing he need specifically clean up, he "Quits."

```
        <200> RIC 3.6              Sunday  November 18, 1979     6:49 pm
The normal <300> pack is back!  It is named 'User2'.  We may still have
mysterious hardware problems, but none have cropped up in two weeks.
Protect yourself.  A new editor is on SYS. -- KAL 11/7

Monitor   Line #1   User:Kal                    Packs:<200>User1,<300>User2
>createImage (executive) Exec03 (lines) 10
     => Image 2
>addRegion (executive) e?
Exec02          Exec03          edit            Editor225
 (executive) Edit (image) 15 (lines) 2 (region index)  XXX

>addRegion (executive) edit (image) 15 (lines)  XXX

>addRegion (executive) edit (image) 2 (lines) 15 (region index) 2
>show status
          SuperWindow
        Name    Process User-Name       Subsystem (Process)

        Monitor 215
        Stager  216
        Exec02  223     edit            Editor (225)
        Exec03  227     compile         TenTelnet (231)

>quit [Confirm]_
```

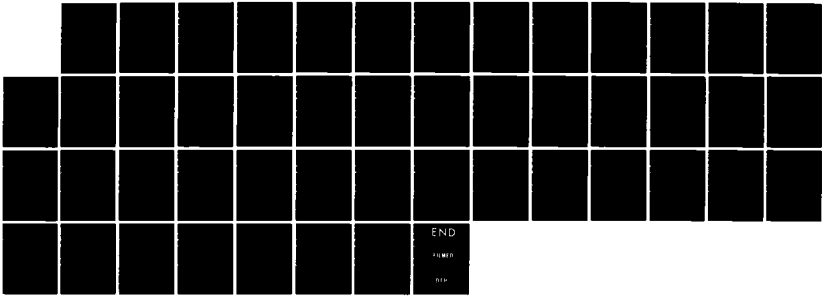
Figure 18. Finishing up.

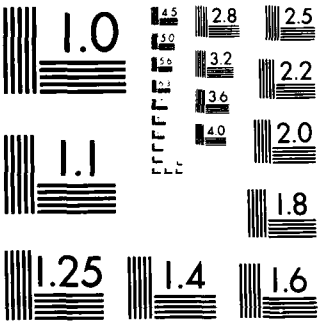
The Monitor tells the Process Manager to kill its tree of processes. Each process in the tree is sent an emergency message asking it to kill itself (in post-order). As each process dies, all interested (registered) parties are notified by emergency message. Thus, if a process fails to clean up its own state -- fails to close its open connections, for example -- the emergency notifications will enable servers to release resources allocated to the dead processes.

When the Monitor dies, the Job Manager is notified. It reinitializes the terminal and waits for the next user.

AD-A155 111 UNIFORM INTERFACES FOR DISTRIBUTED SYSTEMS(U) ROCHESTER 2/2
UNIV NY DEPT OF COMPUTER SCIENCE K A LANTZ MAY 80
TR-63 N00014-78-C-0164

UNCLASSIFIED F/G 9/2 NL





CHAPTER 8

"The Past Through Tomorrow"

RIGITS was designed to provide a distributed system encompassing all available computers. First, we wanted to provide the user with uniform access to all the tools at his disposal, that is, with a single, coherent computing environment that he could tailor to his own preferences and needs. Second, we wanted application programs to be able to communicate with all computers in a consistent manner, that is, processes communicating via messages.

The modularity and flexibility offered by a message-based approach to network communication are an enormous aid in the construction of distributed systems. The introduction of additional system capabilities is easily handled — a process is defined that can perform the new functions and interface to the rest of the system by well-defined interfaces. The dynamic nature of name-service allows new machines, processes, resources and services to be introduced to a running system.

Much of RIGITS has yet to be implemented. I will not repeat each point here (see Chapter 7), but emphasize areas of particular concern.

8.1 Virtual Terminal Management

VTMS is an attempt to provide relatively direct, but sophisticated, access to computing power. It was designed under the assumption that users should be able to communicate with the computer systems at their disposal in as easy and natural a fashion as possible. On the other hand, VTMS facilities must be readily available to application programs.

VTMS has been well received by both sophisticated and unsophisticated users. Although the underlying system contains a number of abstractions that can be difficult for a non-computer scientist to understand, the way in which the system is used to perform multiple tasks and manage screen space is easily demonstrated. One drawback of the current implementation is that VTMS requires a large keyboard with many special function keys (Appendix B). It is not easily adapted to standard terminals. A further factor in the utility of VTMS is the time required to refresh the screen completely; the shorter the time, the more apt the user is to manipulate his screen whenever his display needs

to be changed. A 9600 baud terminal is adequate; 1200 baud terminals cannot easily be used.

We are attempting to add several new facilities to VTMS. Different screen management facilities, based on the "sheet of paper" approach (introduced at the Stanford Artificial Intelligence Laboratory, used in Smalltalk, and extended in DLISP) are being implemented for our Altos. The current implementation of the VTC for the Altos consists of Terminal Input and Output Handlers on the Altos, and the remaining components on the Eclipses, demonstrating the advantages of modular decomposition. It will also be possible to offload much of the VTC into the microcomputer installed in our future keyboards -- similar to the Line Processor for NLS [6]. It might also be possible to further extend VTMS to handle audio/video by extending the definitions of virtual input and output (see [15, 33, 59]).

Graphical input/output has been excluded from VTMS for reasons cited by Irby [105]. There are, however, no fundamental inconsistencies that would prevent its inclusion. The Terminal Input Handler could be extended to allow, for example, the use of pick devices, keysets, and menus. On the other hand, a different Input Handler could be associated with each input device, which would allow the user to specify different types of input simultaneously (more than one Line would be active -- see Section 3.1.1). In either case, the data sent to the Line Handler would have to be marked appropriately. Conversely, the Terminal Output Handler could be extended to handle all modes of output, or a different Output Handler could be associated with each attached output device. However, to include the Pad in the graphic output loop would require a great deal of re-design. (See [56, 85, 160, 201, 202, 218, 222].)

Graphical input/output implies that it may be useful for some tools to gain more direct control of the Terminal. For example, when talking to a remote system via a TELNET-like protocol it may be possible to use a full-screen editor; since the remote system does not know about Virtual Terminals, it assumes it has control of the terminal. The user can easily create an Image containing only the desired tool, but unless the Virtual Terminal Controller is circumvented, control characters will still be fielded locally. One possible solution is the introduction of "transparent" data about which the VTC makes no further assumptions. Another implication is that it may be necessary after all to allow processes to find out if and how their Virtual Terminals are mapped to the screen, contradicting the second "law" of Virtual Terminal management cited in Chapter 3.

Although Terminal Profiles currently allow an application program to tailor its actions on the basis of the available terminal, negotiation of terminal options might also prove useful. Rather than developing separate, but similar, tools for different terminals, negotiation would lead to the development of tools with augmented or restricted capabilities dependent on the terminal. It should also be possible to augment autoblock mode (see Section 3.1.2) to allow the user to specify the amount of output a program may produce before it blocks. Some users would also like to see exactly what they type before the system notifies them of errors; this might be achieved by echoing input

immediately in "light face" (for example) and overwriting that input in "bold face" when the system interprets the input. Negotiation of options is a standard feature of network virtual terminal protocols (see, for example, [18]).

8.2 The Command Interface

The principles of command interaction presented in Chapter 3 have proven themselves in actual use. The suggested implementation of the command interface is elegant but potentially expensive. It is unlikely that current minicomputers can adequately support the vast number of ephemeral processes required for even the simplest activity. However, the functional decomposition of interface modules from service modules allows for many alternative implementations.

Further attention should be given to semantic data bases, expert assistance, and user programming. The best work in these areas includes SIGMA [1, 189], ZOG [187], NLS [63], SPEAKEASY [43], IDA [135], and RITA/Exemplary Programming [4, 5, 223, 224, 225]. Several graduate students at Rochester are pursuing theses in these areas [141, 198].

8.3 Resource Management

The process management facilities of RIGITS have proven quite successful. The basic notion of associating each resource with its own server(s) cannot be faulted. Creation of and access to processes by name allows the system to be dynamically reconfigured. Registration facilities provide for event-driven, prompt error recovery.

The advantages of capabilities cannot be overestimated. Capabilities have been employed successfully in several major message-passing systems, notably DEMOS, Roscoe, and DCCS. Hydra also made extensive use of capabilities, as does LLLNOS. In RIG, connections provide a certain amount of protection, but it is the responsibility of each process to guarantee its own integrity. The individual protection features required in each process are redundant and would be better handled by a central protection mechanism (see Section 8.4).

RIGITS has also paid little attention to authentication, access control, and job control. The concept of a single Job Manager per host that provides these services should prove to be a useful compromise between centralized and decentralized control, but much work is needed, particularly in the areas of job control languages, resource selection, process migration, and distributed control [108].

There have been several attempts to provide effective mechanisms for network job control languages (equivalent to the command programming languages of Chapter 4), including work at UCLA [34, 86, 87] and for the European Informatics Network [191]. The National Bureau of Standards is extending the principles of its Network Access Machine into this area

[76, 118]. Other examples include PCL [132], NSL [209], and PACE [3]. General job control issues are discussed in, for example, [78, 156].

With regard to access control and authentication, computer security is an important topic. See, for example, [28, 38, 50, 57, 114, 158, 167, 234].

Sophisticated resource selection and management necessitates the use of distributed data bases. Surveys of current work are contained in [97, 155].

8.4 Exception Handling

Intra-process exception handling, as currently implemented, is insufficient for a sophisticated system. Provisions should be made for associating handlers with particular exceptions, similar to Levin's mechanisms [133] or those of CLU [137].

The emergency message provides a particularly elegant mechanism for asynchronous, inter-process exception handling. As with intra-process exceptions, it should be possible to associate different emergency handlers with different emergencies. Some relevant remarks may be found in [71].

Event handlers and spooling agents will be particularly important in the future. The key problem here is the development of "event languages" that can be used to define events. Particularly difficult problems are negations (what exactly is a non-event) and the cessation of one event while waiting for others to complete.

General issues of fault tolerance and error recovery are discussed in, for example, [55, 101, 177, 195, 196].

8.5 Distributed Computing

> There is not yet agreement on the appropriate mechanisms for parallel programming. This results from the rapid changes in the field, the lack of a widely recognized set of criteria, the immense variety of applications and hardware architectures, and the diversity of philosophies about how systems should be structured. [156, p. 2, quoting G. Andrews]

Formal paradigms for distributed computing include communicating sequential processes [95], Actor systems [92, 93, 94], distributed processes [31], LIMP [103], DREAM [183]. synchronizing resources [10], Flowgraphs [152], PARLANCE [180], and PLITS [71].

Some of the important questions include: How should processing be distributed -- to achieve functional separation, closeness according to some measure, a given level of fault tolerance? How should processes be organized for control and communication? Is the control distributed or centralized? How is synchronization maintained? How should data be distributed -- replicated at each processing node, partitioned so that only some of the data exist at given nodes? How many of the distributed system naming, error control, resource management, and other mechanisms should be visible and under user control or handled automatically?

RIGITS and related work at Rochester has proposed some relatively straightforward answers to some of these questions, but much work remains before a standard can be achieved.

8.6 Software Engineering

Lastly, new advances in software engineering, methodology, and management are required before distributed systems can be built, documented, and maintained with the ease of traditional single-processor systems. Relevant work includes [79, 172, 173, 176, 182, 197, 220, 233].

8.7 Crescat Scientia Vita Excolatur

RIG, the implementation, has been acclaimed and abused, and deserves both. RIGITS, the design, has met with notably few objections. In fact, RIGITS is having considerable impact on such diverse projects as the DARPA/DMA Image Understanding Testbed [16], Carnegie-Mellon's personal computing environment [33], and interprocess communication facilities for UNIX [174]. These systems, as well as new versions of RIG, will further test the concepts of RIGITS and certainly lead to new and better ideas.

Bibliography

1] R.J. Abbott. A command language processor for flexible interface design. ISI/RR-74-24, USC/Information Sciences Institute, September 1974.

2] M.D. Abrams, R.P. Blanc, and I.W. Cotton. Computer Networks: A Tutorial. IEEE Press, revised 1978.

3] T.A. Akin, P.B. Flinn, and D.H. Forsyth Jr. A prototype for an advanced command language. Proc. 16th Annual Southeastern Regional Conference 96-102, ACM, April 1978.

4] R.H. Anderson and J.J. Gillogly. Rand Intelligent Terminal Agent (RITA): Design philosophy. R-1809-ARPA, The Rand Corporation, February 1976.

5] R.H. Anderson and J.J. Gillogly. The Rand Intelligent Terminal Agent (RITA) as a network access aid. Proc. National Computer Conference 45:501-509, AFIPS, June 1976.

6] D.I. Andrews. Line processor -- A device for amplification of display terminal capabilities for text manipulation. Proc. National Computer Conference 43:257-265, AFIPS, June 1974.

7] D.I. Andrews, B.R. Boli, and A.A. Poggio. An introduction to the Frontend. Augmentation Research Center, SRI International, January 1977.

8] D.I. Andrews, B.R. Boli, and A.A. Poggio. A guide to the Command Meta Language and the Command Language Interpreter. Augmentation Research Center, SRI International, January 1977.

9] D.I. Andrews, L.L. Garlick, and A.A. Poggio. Frontend system documentation. Augmentation Research Center, SRI International, January 1977.

10] G. Andrews. Synchronizing resources. TR 78-360, Computer Science Department, Cornell University, February 1979.

11] J.L Baer. A survey of some theoretical aspects of multiprocessing. Computing Surveys 5(1):31-80, March 1973.

12] J.E. Ball, E.J. Burke, I. Gertner, K.A. Lantz, and R.F. Rashid. Perspectives on message-based distributed computing. Proc. Computer Networking Symposium 46-51, NBS/IEEE, December 1979. Also TR43, Computer Science Department, University of Rochester.

13] J.E. Ball, J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner. RIG, Rochester's Intelligent Gateway: System overview. IEEE Transactions on Software Engineering SE-2(4):321-328, December 1976. Also TR5, Computer Science Department, University of

Rochester.

14] J.E. Ball, G. Williams, and J.R. Low. A preliminary ZENO language description. TR41, Computer Science Department, University of Rochester, January 1979.

15] D.L.A. Barber. The real virtual terminal. EIN/DLAB/REALVTP, European Informatics Network, February 1977. Also INWG Protocol Note 64, IFIP International Network Working Group 6.1.

16] H.G. Barrow, E. Fischler, G. Jirak, D. Kashtan, L. Quam, and J. Tenenbaum. ARPA/DMA IU Automated Cartography Testbed Working Paper. Artificial Intelligence Center, SRI International, October 1979

17] F. Baskett, J.H. Howard, and J.T. Montague. Task communication in DEMOS. Proc. 6th Symposium on Operating Systems Principles 23-32, ACM, published as SIGOPS Operating Systems Review 11(5), November 1977.

18] E. Bauwens and F. Magnee. Remarks on negotiation mechanism and attention handling. SART 77/12/13, Department of Systems and Automatic Control, University of Liege, May 1977. Also INWG Protocol Note 72, IFIP International Network Working Group 6.1.

19] E. Bauwens and F. Magnee. The virtual terminal approach in the Belgian University Network. Computer Networks 2(4/5):297-311, September/October 1978.

20] R. Bayer, R.M. Graham, and G. Seegmueller, editors. Operating Systems: An Advanced Course. Springer-Verlag, 1978.

21] Bell Laboratories. The Bell System Technical Journal 57(6), special issue on UNIX, July/August 1978.

22] J.W. Benoit and E. Graf-Webster. Evolution of network user services -- The Network Resource Manager. Proc. Symposium on Computer Networks: Trends and Applications 21-24, NBS/IEEE, May 1974.

23] R.P. Blanc and I.W. Cotton, editors. Computer Networking. IEEE Press, 1976.

24] D.G. Bobrow, J.D. Burchfiel, D.L. Murphy, and R.S. Tomlinson. TENEX, a paged time sharing system for the DEC-10. CACM 15(3):135-143, March 1972.

25] D.R. Boggs, J.F. Shoch, E.A. Taft, and R.M. Metcalfe. Pup: An internetwork architecture. CSL-79-10, Xerox Palo Alto Research Center, July 1979. To appear in IEEE Transactions on Communications, April 1980.

26] J.H. Botterill and W.O. Evans. The rule-driven Control Language for System/38. In IBM System/38 Technical Developments, pages 83-86. IBM General Systems Division, 1978.

27] W.J. Bouknight, G.R. Grossman, and D.M. Grothe. The ARPA Network Terminal System -- A new approach to network access. Proc. 3rd Data Communications Symposium 73-79, ACM/IEEE, November 1973.

28] D.K. Branstad, editor. Computer security and the data encryption standard. Special Publication 500-27, National Bureau of Standards, February 1978.

29] P. Brinch Hansen. The nucleus of a multiprogramming system. CACM 13(4):238-241+, April 1970.

30] P. Brinch Hansen. Operating Systems Principles. Prentice Hall, 1973.

31] P. Brinch Hansen. Distributed processes: A concurrent programming language. CACM 21(11):934-941, November 1978.

32] E.J. Burke. AMPS: The Alto Message-Passing System. Internal Memo, Computer Science Department, University of Rochester, April 1980.

33] Carnegie-Mellon University Computer Science Department. Proposal for a joint effort in personal scientific computing. August 1979.

34] C. Carper, T.E. Gray, and C. Lai. A universal job control language. UCLA-ENG-7544, Computer Science Department, University of California-Los Angeles, July 1975.

35] P.M. Cashin. Datapac network protocols. Proc. 3rd International Conference on Computer Communications 150-155, International Council on Computer Communications, August 1976.

36] V.G. Cerf and P.T. Kirstein. Issues in packet-network interconnection. Proc. IEEE 66(11):1386-1408, November 1978.

37] K.M. Chandy and J. Misra. Deadlock absence proofs for networks of communicating processes. Information Processing Letters 9(4), November 1979.

38] P.L. Chaum and R.S. Fabry. Implementing capability-based protection using encryption. Memorandum UCB/ERL M78/46, Electronics Research Laboratory, University of California - Berkeley, July 1978.

39] D.C. Cheriton. Multi-process Structuring and the Thoth Operating System. Ph.D. thesis, University of Waterloo, 1978. Also Technical Report, University of British Columbia, March 1979.

40] D.C. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. Thoth, a portable real-time operating system. CACM 22(2):105-115, February 1979.

41] G.L. Chesson. The network UNIX system. Proc. 5th Symposium on Operating Systems Principles 60-66, ACM, published as SIGOPS

Operating Systems Review 9(5), November 1975.

42] E. Cohen and D. Jefferson. Protection in the Hydra operating system. Proc. 5th Symposium on Operating Systems Principles 141-160, ACM, published as SIGOPS Operating Systems Review 9(5), November 1975.

43] S. Cohen and Pieper, editors. SPEAKEASY-3 Reference Manual Level Lambda IBM OS/VS Version. Argonne National Laboratories, 1976.

44] G.E. Conant and S. Wecker. DNA: An architecture for heterogeneous computer networks. Proc. 3rd International Conference on Computer Communications 618-625, International Council on Computer Communications, August 1976.

45] R.P. Cook. *MOD — A language for distributed programming. Proc. 1st International Conference on Distributed Computing Systems 233-241, IEEE, October 1979.

46] B.P. Cosell, P.R. Johnson, J.H. Malman, R.E. Schantz, J. Sussman, R.H. Thomas, and D.C. Walden. An operational system for computer resource sharing. Proc. 5th Symposium on Operating Systems Principles 75-81, ACM, published as SIGOPS Operating Systems Review 9(5), November 1975.

47] S.D. Crocker, J.F. Heafner, R.M. Metcalfe, and J.B. Postel. Function-oriented protocols for the ARPA computer network. Proc. Spring Joint Computer Conference 40:271-279, AFIPS, May 1972. Reprinted in Blanc and Cotton [23], pages 86-94.

48] A. Danet, R. Despres, A. Le Rest, G. Pichon, and S. Ritzenthaler. The French public packet switching service: The Transpac network. Proc. 3rd International Conference on Computer Communications 251-259, International Council on Computer Communications, August 1976.

49] J. Davidson, W. Hathaway, J. Postel, N. Mimno, R. Thomas, and D. Walden, The ARPANET TELNET protocol: Its purpose, principles, implementation, and impact on host operating system design. Proc. 5th Data Communications Symposium 4.10-4.18, ACM/IEEE, September 1977.

50] D.W. Davies. Protection. To appear in Lampson [121].

51] D.W. Davies and D.L.A. Barber. Communications Networks for Computers. John Wiley and Sons, 1973.

52] D.W. Davies, D.L.A. Barber, W.L. Price, and C.M. Solomonides. Computer Networks and their Protocols. John Wiley and Sons, 1979.

53] J. Day. TELNET Data Entry Terminal Option. ARPA Network Working Group RFC 732, Network Information Center, SRI International, September 1977.

54] J. Day. Resource sharing protocols. Computer 12(9):47-56, September 1979.

55] P. Denning. Fault-tolerant operating systems. Computer Surveys 8(4):361-386, December 1976.

56] P. Denning, editor. Computing Surveys 10(4), special issue on graphics standards, December 1978.

57] P. Denning, editor. Computing Surveys 11(4), special issue on cryptology, December 1979.

58] R. des Jardins and G. White. ANSI reference model for distributed systems. Proc. Fall COMPCON 144-149, IEEE, September 1978.

59] W.C. Donelson. Spatial management of information. SIGGRAPH '78 Proceedings 203-209, published as Computer Graphics 12(3), Summer 1978.

60] J.E. Donnelley. A distributed capability computing system. Proc. 3rd International Conference on Computer Communications 432-440, International Council on Computer Communications, August 1976.

61] J.E. Donnelley. Components of a networks operating system. Proc. 4th Conference on Local Computer Networks 1-12, IEEE, October 1979.

62] D. Eastlake. ITS status report. AI Memo 238, MIT Artificial Intelligence Laboratory, April 1972.

63] D.C. Engelbart. Knowledge workshop development. Final Report, SRI Project 1868, Augmentation Research Center, Stanford Research Institute, January 1976.

64] D.C. Engelbart and W.K. English. A research center for augmenting human intellect. Proc. Fall Joint Computer Conference 33:395-410, AFIPS, September 1968.

65] EURONET Working Group on Standardization. Data-entry virtual terminal for Euronet. VTP-D/4, Commission of the European Community, November 1978. Also TER 051.3, Reseau Cyclades.

66] R.A. Faneuf. The National Software Works: Operational issues in a distributed processing system. Proc. National Conference 39-43, ACM, October 1977. Also CA-7708-1911, Massachusetts Computer Associates.

67] D.J. Farber, J. Feldman, F.R. Heinrich, M.D. Hopwood, K.C. Larson, D.C. Loomis, and L.A. Rowe. The Distributed Computing System. Proc. COMPCON 31-34, IEEE, March 1973.

68] D.J. Farber and F.R. Heinrich. The structure of a distributed computing system -- The distributed file system. Proc. 1st International Conference on Computer Communications 364-370, ACM/IEEE, October 1972.

69] D.J. Farber and K.C. Larson. The structure of a distributed computing system -- Software. Proc. Computer-Communications Networks and Teletraffic 539-545, April 1972.

70] D.J. Farber and K.C. Larson. The system architecture of the distributed computer system -- The communications system. Proc. Computer-Communications Networks and Teletraffic 21-27, April 1972.

71] J.A. Feldman. High-level programming for distributed computing. CACM 22(6):363-368, June 1979.

72] J.A. Feldman, J.R. Low, and P.D. Rovner. Programming distributed systems. Proc. Annual Conference 310-316, ACM, December 1978.

73] J.A. Feldman and A. Nigam. A model and proof technique for message-based systems. To appear in SIAM Journal of Computing, 1980.

74] J.A. Feldman and R.F. Rashid. System support for a distributed image understanding program. Proc. Image Understanding Workshop 83-86, DARPA, April 1977.

75] J.A. Feldman and R.F. Sproull. System support for the Stanford Hand-Eye System. Proc. 2nd International Joint Conference on Artificial Intelligence 183-189, International Joint Council on Artificial Intelligence, September 1971.

76] M.L. Fitzgerald. Common command language for file manipulation and network job execution. Special Publication 500-37, National Bureau of Standards, August 1978.

77] J.G. Fletcher and R.W. Watson. Service support for network operating system services. Proc. Spring COMPCON 415-424, IEEE, February 1980.

78] N. Francez. On achieving distributed termination. In G. Kahn, editor, Semantics of Concurrent Computation: Proceedings of the International Symposium, pages 300-315. Springer-Verlag, 1979.

79] M. Freeman, W.W. Jacobs, and L.S. Levy. On the construction of interactive systems. Proc. National Computer Conference 47:555-562, AFIPS, June 1978.

80] GEC Computers Limited. GEC 4000 series computers: GEC 4070, 4080, 4082 technical description. Elstree Way, Borehamwood, Hertfordshire, England, November 1976.

81] D.P. Geller. The National Software Works -- Access to distributed files and tools. Proc. National Conference 53-58, ACM, October 1977. Also CA-7708-1111, Massachusetts Computer Associates.

82] I. Gertner. Performance evaluation of communicating processes. Proc. Conference on Simulation, Modeling, and Measurement of

Computer Systems 241-249, ACM, August 1979.

83] I. Gertner. Performance Evaluation of Communicating Processes. Ph.D. thesis, University of Rochester, 1980.

84] J.B. Goodenough. Exception handling: Issues and a proposed notation. CACM 18(12):683-696, December 1975.

85] Graphics Standards Planning Committee. Status report. Computer Graphics 13(3):I.1-V.10, August 1979.

86] T.E. Gray. Job control in a network computing environment. Proc. Fall COMPCON 146-149, IEEE, February 1976.

87] T.E. Gray and G. Estrin. Models for network job control. Internal Memorandum 170, Computer Science Department, University of California - Los Angeles, March 1977.

88] J.F. Heafner. A methodology for selecting and refining man-computer languages to improve users' performance. ISI/RR-74-21, USC/Information Sciences Institute, September 1974.

89] J.F. Heafner. Protocol analysis of man-computer languages: Design and preliminary findings. ISI/RR-75-34, USC/Information Sciences Institute, July 1975.

90] G.G. Hendrix. The LIFER manual: A guide to building practical natural language interfaces. Technical Note 138, Artificial Intelligence Center, SRI International, February 1977.

91] G.G. Hendrix. Human engineering for applied natural language processing. Technical Note 139, Artificial Intelligence Center, SRI International, February 1977.

92] C. Hewitt and H.G. Baker. Laws for communicating parallel processes. Proc. IFIP Congress 77 987-992, IFIP, August 1977.

93] C. Hewitt, G. Attardi, and H. Lieberman. Specifying and proving properties of guardians for distributed systems. In G. Kahn, [illegible]or, Semantics of Concurrent Computation: Proceedings of the International Symposium, pages 316-336. Springer-Verlag, 1979.

94] C. Hewitt, G. Attardi, and H. Lieberman. Security and modularity in message passing. Proc. 1st International Conference on Distributed Computing Systems 347-358, IEEE, October 1979.

95] C.A.R. Hoare. Communicating sequential processes. CACM 21(8):666-677, August 1978.

96] E. Holler. Case study: The NSW. To appear in Lampson [121].

97] E. Holler. Multiple copy update. To appear in Lampson [121].

```
     message = pointer to a message
          (default:  @PGL)

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

  => response = SENDOK;
          = SENDFAILED, send failed
          = SENDWITHSECONDQ, message placed on secondary queue
```

A.1.2 Port Management

```
HowMany ([port]) --> messagecount

          Return the number of messages waiting on a  given  port  if  a
          port  is  specified.   If no port is specified then return the
          total number of messages waiting for the current process.

     port = port to check
          (default:  check all ports)

  => messagecount = number of messages  waiting  on  specified  port  or
          total number of waiting messages


IsOpenPort (port) --> open

          Check to see if a port is open (i.e.  not locked).

     port = port to check

  => open = true, if the port is open;
          = false, otherwise


LockUp (port)

          Lock a port so that no more message will be accepted from  it.
          Incoming message will still be queued, however.

     port = port to lock


NumPorts () --> numports

          Return number of ports possessed by current process.
```

```
Send ([message, dontwait, dontfree]) --> response

          Send a message from one process to another.

     message = pointer to message (default:  @PGL)

     dontwait = false --> suspend the sender if the receiver's  port  is
          full;
          = true --> don't suspend the sender
          (default:  false)

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

  => response = true, if everything ok;
          = false, if dontwait was true and message could not be sent


SendAck ([message, timeout, dontfree]) --> messageacked

          Send a message to a process-port and wait for an acknowledging
          message  from the same process-port.  The acknowledgement must
          be sent to the same port used to send  the  original  message.
          The  acknowledgement  message  will be contained in the memory
          block pointed to by message.

     message = pointer to a message
          (default:  @PGL)

     timeout = number of seconds to wait for a reply before giving up
          (default:  wait forever)

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

  => messageacked = true, acknowledgement received;
          = false, timeout occurred


SendDontWait ([message, dontfree]) --> response

          Send a message.  If the receiver's port  is  full,  place  the
          message  on a secondary queue of message.  Notify sender via a
          system message when the message is finally put on the  primary
          queue  for  that  port.  Note that it is an error for a single
          process to place more than one message on the secondary  queue
          of another process.
```

port will be received.

message = pointer to a message; MyPort field must contain a valid port number for receiving a message
(default: @PGL)

timeout = number of seconds to wait for a message before timing out
(default: wait forever)

=> messagereceived = true, message received and placed in memory block pointed to by "message;"
= false, timeout has occurred

ReceiveAny ([message, timeout]) —> messagereceived

Receive a message on any port.

message = pointer to a message
(default: @PGL)

timeout = number of seconds to wait for a message
(default: wait forever)

=> messagereceived = true, message received and packed in memory block pointed to by "message;"
= false, timeout has occurred

ReceiveSpecific ([message, timeout]) —> messagereceived

Receive a message from a particular sending process-port which has been sent to a particular port of the calling process. ReceiveSpecific can be used in conjunction with Send to simulate SendAck.

message = pointer to a message; the OtherProc and OtherPort fields of message must refer to a valid process-port; the MyPort field of message must refer to a valid port of the calling process
(default: @PGL)

timeout = number of seconds to wait for a message
(default: wait forever)

=> messagereceived = true, message received and packed in memory block pointed to by "message;"
= false, timeout has occurred

APPENDIX A

Message Primitives

The routines presented here are variants on those currently used in RIG. The first line of each routine contains the calling sequence, with optional arguments contained in square brackets, and results indicated by "-->". In the list of results, "@<parameter>" means the associated parameter is a pointer, and the routine modifies the contents of the location pointed to by that parameter. Following this definition is a discussion of what the routine does, then the definition of the arguments and results, demarcated by "=>".

@PGL refers to the "process globals" associated with each process. They are initialized by the Kernel when the process is created and define the "working environment" of the process.

A.1 Kernel Calls

A.1.1 Message-passing

PriorityReceive (prioritylist [, message, timeout]) --> messagereceived

Receive a message on any port but in the order specified by a priority list of port numbers. The message received is determined by "prioritylist" in the sense that it will come from the port of highest priority which has a message at the time of the call. If no message is present, the first message received during the timeout period will be returned.

prioritylist = a list of port numbers

message = pointer to a message
(default: @PGL)

timeout = number of seconds to wait for a message
(default: wait forever)

=> messagereceived = true, message received and packed in memory block pointed to by "message;"
= false, timeout has occurred

Receive ([message, timeout]) --> messagereceived

Receive a message on a particular port (message>>Message.MyPort). Only messages destined for that

231] J.E. White. Elements of a distributed programming system. ARPA Network Working Group RFC 708, Network Information Center, SRI International, January 1976.

232] J.E. White. A high-level framework for network-based resource sharing. Proc. National Computer Conference 45:561-570, AFIPS, June 1976.

233] D. Wilczynski, R. Tugender, and D. Oestreicher. The SIGMA experience: A study in the evolutionary design of a large software system. Proc. National Computer Conference 48:847-853, AFIPS, June 1979.

234] H.M. Wood and S.R. Kimbleton. Access control mechanisms for a network operating system. Proc. National Computer Conference 48:821-829, AFIPS, June 1979.

235] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. CACM 17(6):, June 1974.

236] H. Zimmermann. The ISO reference model. To appear in IEEE Transactions on Communications, April 1980.

216] S. Treu. Required mental work as a criterion for interactive command language design. Technical Report 73-11, Computer Science Department, University of Pittsburgh, August 1973.

217] C. Unger, editor. Command Languages. North-Holland, 1975.

218] J. van den Bos. Definition and use of higher-level graphics input tools. SIGGRAPH '78 Proceedings 38-42, published as Computer Graphics 12(3), August 1978.

219] A. van Lamsweerde and M. Sintzoff. Formal derivation of strongly correct concurrent programs. Acta Informatica 12:1-31, 1979.

220] K.E. Victor. A software engineering environment. Proc. Computers in Aerospace Conference 399-403, AIAA/NASA/IEEE/ACM, November 1977.

221] D.C. Walden. A system for interprocess communication in a resource-sharing computer network. CACM 15(4):221-230, April 1972.

222] V.L. Wallace. The semantics of graphic input devices. Proc. Symposium on Graphic Languages 61-65, ACM, published as Computer Graphics 10(1), Spring 1976.

223] D.A. Waterman. Exemplary programming in RITA. P-5861, The Rand Corporation, October 1977.

224] D.A. Waterman. Rule-directed interactive transaction agents: An approach to knowledge acquisition. R-2171-ARPA, The Rand Corporation, February 1978.

225] D.A. Waterman. A rule-based approach to knowledge acquisition for man-machine interface programs. P-5823, The Rand Corporation, June 1978.

226] R.W. Watson. User interface design issues for a large interactive system. Proc. National Computer Conference 45:357-364, AFIPS, June 1976.

227] R.W. Watson. Network architecture design for back-end storage networks. Computer 13(2):32-48, February 1980.

228] R.W. Watson. Distributed system architecture model. To appear in Lampson [121].

229] R.W. Watson. Naming in distributed systems. To appear in Lampson [121].

230] R.W. Watson and J.G. Fletcher. An architecture for support of network operating system services. Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks 18-50, Lawrence Berkeley Laboratory, August 1979.

1979.

202] R.F. Sproull and E.L. Thomas. A network graphics protocol. Computer Graphics 8(3), Fall 1974.

203] R. Stotz, P. Raveling, and J. Rothenburg. The terminal for the Military Message Experiment. Proc. National Computer Conference 48:855-861, AFIPS, June 1979.

204] R. Stotz, R. Tugender, D. Wilczynski, and D. Oestreicher. SIGMA -- An interactive message service for the Military Message Experiment. Proc. National Computer Conference 48:839-846, AFIPS, June 1979.

205] H. Sturgis. A Postmortem for a Time Sharing System. Ph.D. thesis, University of California - Berkeley, 1973.

206] C.A. Sunshine. Interconnection of computer networks. Computer Networks 1(3):175-195, January 1977.

207] D.C. Swinehart. COPILOT: A Multiple Process Approach to Interactive Programming Systems. Ph.D. thesis, Stanford University, 1974.

208] D.C. Swinehart, G. McDaniel, and D. Boggs. WFS: A simple shared file system for a distributed environment. Proc. 7th Symposium on Operating Systems Principles 9-17, ACM, December 1979.

209] F. Tarini, R. Sharp, M. Martelli, and A. Endrizzi. A network systems language. Proc. 1st International Conference on Distributed Computing Systems 305-314, IEEE, October 1979.

210] W. Teitelman. A display oriented programmer's assistant. CSL 77-3, Xerox Palo Alto Research Center, March 1977.

211] W. Teitelman, et al. INTERLISP Reference Manual. Xerox Palo Alto Research Center, 3rd revision December 1978.

212] C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull, and D.R. Boggs. Alto: A personal computer. To appear in D. Siewiorek, C.G. Bell, and A. Newell, editors, Computer Structures: Readings and Examples. McGraw-Hill, second edition in press.

213] R.H. Thomas. A Resource Sharing Executive for the ARPANET. Proc. National Computer Conference 42:155-163, AFIPS, June 1973.

214] R.H. Thomas, R.E. Schantz, and H.C. Forsdick. Network operating systems. Report No. 3796, Bolt Beranek and Newman, March 1978.

215] K.J. Thurber, editor. Tutorial: Distributed Processor Communication Architecture. IEEE Press, 1979.

187] G. Robertson, A. Newell, and K. Ramakrishna. ZOG: A man-machine communication philosophy. Computer Science Department, Carnegie-Mellon University, August 1977.

188] R. Rosenthal and B.D. Lucas. The design and implementation of the National Bureau of Standards' Network Access Machine (NAM). Special Publication 500-35, National Bureau of Standards, June 1978.

189] J. Rothenburg. An intelligent tutor: On-line documentation and help for a military message service. ISI/RR-74-26, USC/Information Sciences Institute, May 1975.

190] J.H. Saltzer. Naming and binding of objects. In Bayer, Graham, and Seegmueller [20], pages 99-208.

191] P. Schicker and A. Duenki. Network job control and its supporting services. Proc. 3rd International Conference on Computer Communications 303-307, International Council on Computer Communications, August 1976.

192] P. Schicker and A. Duenki. The virtual terminal definition. Computer Networks 2(6):429-441, December 1978.

193] P. Schicker and H. Zimmermann. Proposal for a scroll mode virtual terminal. SIGCOMM Computer Communications Review 7(3):23-55, July 1977.

194] M. Schroeder. Cooperation of Mutually Suspicious Subsystems. Ph.D. thesis, Massachusetts Institute of Technology, 1972.

195] R.M. Shapiro and R.E. Millstein. The NSW reliability plan. CA-7701-1411, Massachusetts Computer Associates, June 1977.

196] R.M. Shapiro and R.E. Millstein. Reliability and fault recovery in distributed processing. Oceans '77 Conference Record 2:31D.1-31D.5, October 1977.

197] B. Shneiderman. Human factors experiments in designing interactive systems. Computer 12(2):9-19, December 1979.

198] E.T. Smith. Ph.D. thesis, University of Rochester, in preparation April 1980.

199] M.H. Solomon and R.A. Finkel. Roscoe: A multi-microcomputer operating system. Report 321, Computer Sciences Department, University of Wisconsin - Madison, May 1978.

200] M.H. Solomon and R.A. Finkel. The Roscoe distributed computing system. Proc. 7th Symposium on Operating Systems Principles 108-114, ACM, December 1979.

201] R. F. Sproull. Raster graphics for interactive programming environments. CSL-79-6, Xerox Palo Alto Research Center, July

47:953-966, AFIPS, June 1978.

173] C.V. Ramamoorthy and R.T. Yeh. Tutorial: Software Methodology. IEEE Press, 1978.

174] R.F. Rashid. An interprocess communication facility for UNIX. Computer Science Department, Carnegie-Mellon University, February 1980.

175] D.D. Redell, Y.K. Dalal, T.R. Horsley, H.C. Lauer, W.C. Lynch, P.R. McJones, H.G. Murray, and S.C. Purcell. Pilot: An operating systems for a personal computer. CACM 23(2):81-92, February 1980.

176] D.J. Reifer, editor. Tutorial: Software Management. IEEE Press, 1980.

177] D.A. Rennels. Distributed fault-tolerant computer systems. Computer 13(3):55-65, March 1980.

178] D.L. Retz. ELF: A system for network access. Intercon Conference Record 25/2:1-5, IEEE, April 1975.

179] D.L. Retz and B.W. Schafer. Structure of the ELF operating system. Proc. National Computer Conference 45:1007-1016, AFIPS, June 1976.

180] P. Reynolds. Parallel Processing Structures: Languages, Schedules, and Performance Results. Ph.D. thesis, University of Texas - Austin, 1979.

181] M. Richards. BCPL: A tool for compiler writing and systems programming. Proc. Spring Joint Computer Conference 34:557-566, AFIPS, May 1969.

182] W.E. Riddle and R.E. Fairley. Review of the Pingree Park software development tools workshop. Proc. Computers in Aerospace Conference II, October 1979.

183] W.E. Riddle, J.H. Sayler, A.R. Segal, A.M. Stavely, and J.C. Wileden. A description scheme to aid the design of collections of concurrent processes. Proc. National Computer Conference 47:549-554, AFIPS, June 1978.

184] RIG Working Group. RIG Implementer's Guide. Internal Memo, Computer Science Department, University of Rochester, 1979.

185] RIG Working Group. RIG Reference Manual. Internal Memo, Computer Science Department, University of Rochester, 1979.

186] L.G. Roberts and B.D. Wessler. Computer network development to achieve resource sharing. Proc. Spring Joint Computer Conference 36:543-549, AFIPS, May 1970.

157] N. Naffah. High level protocol for alphanumeric data-entry terminals. Computer Networks 2(2):84-94, May 1978.

158] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. CACM 21(12):993-998, December 1978.

159] A.J. Nemeth. Bolt Beranek and Newman Inc. Personal communication, July 1979.

160] W.M. Newman and R.F. Sproull. Principles of Interactive Computer Graphics. McGraw-Hill, second edition 1979.

161] A. Nigam. A Specification and Verification Formalism for Distributed Database Algorithms. Ph.D. thesis, University of Rochester, in preparation, April 1980.

162] NSW Protocol Committee. MSG: The interprocess communication facility for the National Software Works. Report No. 3483, Bolt Beranek and Newman, December 1976. Also CADD-7612-2411, Massachusetts Computer Associates.

163] J.K. Ousterhout, D.A. Scelze, and P.S. Sindhu. Medusa: An experiment in distributed operating system design. CACM 23(2):92-105, February 1980.

164] D.L. Parnas. On the criteria to be used in decomposing systems into modules. CACM 15(12):1053-1058, December 1972.

165] D.L. Parnas. On a buzzword: Hierarchical structure. Proc. IFIP Congress 74 336-339, IFIP, August 1974.

166] W.H. Paxton. A client-based transaction system to maintain data integrity. Proc. 7th Symposium on Operating Systems Principles 18-23, ACM, December 1979.

167] G.J. Popek. Security in network operating systems: A survey. Interim Report, National Bureau of Standards, July 1978.

168] J.B. Postel. INTERNET protocol. IEN 111, Defense Advanced Research Projects Agency, August 1979.

169] J.B. Postel. Transmission control protocol. IEN 112, Defense Advance Research Projects Agency, August 1979.

170] L. Pouzin. Presentation and major design aspects of the Cyclades computer network. Proc. 3rd Data Communications Symposium 80-87, ACM/IEEE, November 1973.

171] L. Pouzin and H. Zimmermann. A tutorial on protocols. Proc. IEEE 66(11):1346-1370, November 1978.

172] C.V. Ramamoorthy and G.S. Ho. A design methodology for user oriented computer systems. Proc. National Computer Conference

142] F. Magnee, A. Endrizzi, and J. Day. A survey of terminal protocols. SART 79/03/02.1, Department of Systems and Automatic Control, University of Liege, June 1979.

143] W.C. Mann. Man-machine communication research: Final report. ISI/RR-77-57, USC/Information Sciences Institute, February 1977.

144] E.G. Manning, R. Howard, C.G. O'Donnell, K. Pammett, and E. Chang. A UNIX-based local processor and network access machine. Computer Networks 1(2):139-142, September 1976.

145] E.G. Manning and R.W. Peebles. A homogeneous network for data-sharing communications. Computer Networks 1(4):211-224, May 1977.

146] M.P. Mariani and D.F. Palmer, editors. Tutorial: Distributed System Design. IEEE Press, 1979.

147] J.M. McCrossin, R.P. O'Hara, and L.R. Koster. A time-sharing display terminal session manager. IBM System Journal 17(3):260-275, 1978.

148] J.M. Mcquillan and V.G. Cerf. Tutorial: A Practical View of Computer Communication Protocols. IEEE Press, 1978.

149] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. CACM 19(7):395-403, July 1976.

150] D.L. Mills. An overview of the Distributed Computer Network. Proc. National Computer Conference 45:523-531, AFIPS, June 1976.

151] R.E. Millstein. The National Software Works: A distributed processing system. Proc. National Conference 44-52, ACM, October 1977.

152] G. Milne and R. Milner. Concurrent processes and their syntax. JACM 26(2):302-321, April 1979.

153] J.G. Mitchell, J. Newcomer, A. Perlis, H. van Zoeren, and D. Wile. Conversational programming: LCC. Computer Science Department, Carnegie-Mellon University, June 1971.

154] M.L. Model. Monitoring System Behavior in a Complex Computational Environment. Ph.D. thesis, Stanford University, 1979.

155] C. Mohan. Distributed data base management: Progress, problems, some proposals and future directions. Working Paper WP-7802, Computer Science Department, University of Texas - Austin, May 1979.

156] C. Mohan. A perspective on distributed computing models, constructs, methodologies, and applications. Working Paper DSG-8001, Computer Science Department, University of Texas - Austin, January 1980.

Rochester, May 1979.

129] K.A. Lantz and R.F. Rashid. Virtual terminal management in a multiple process environment. Proc. 7th Symposium on Operating Systems Principles 86-97, ACM, December 1979.

130] H.C. Lauer and R.M. Needham. On the duality of operating system structures. Proc. 2nd International Symposium on Operating Systems, IRIA, October 1978. Reprinted in SIGOPS Operating Systems Review 13(2):3-19, April 1979.

131] V.R. Lesser and D.D. Corkill. Functionally-accurate cooperative distributed systems. COINS Technical Report 79-12, University of Massachusetts - Amherst, February 1979.

132] V.R. Lesser, D. Serrain, and J. Bonar. PCL: A process-oriented job control language. Proc. 1st International Conference on Distributed Computing Systems 315-329, IEEE, October 1979. Also COINS Technical Report 78-22, University of Massachusetts - Amherst, November 1978.

133] R. Levin. Program Structures for Exceptional Condition Handling. Ph.D. thesis, Carnegie-Mellon University, 1977.

134] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. Proc. 5th Symposium on Operating Systems Principles 132-140, ACM, published as SIGOPS Operating Systems Review 9(5), November 1975.

135] R.F. Ling and H.V. Roberts. User's Manual for IDA. The Scientific Press, Palo Alto, California, 1980.

136] B. Liskov. Primitives for distributed computing. Proc. 7th Symposium on Operating Systems Principles 33-42, ACM, December 1979.

137] B. Liskov and A. Snyder. Exception handling in CLU. IEEE Transactions on Software Engineering SE-5(6):546-558, November 1979.

138] J. Livesey. Inter-process communication and naming in the MININET system. Proc. Spring COMPCON 222-229, IEEE, February 1979.

139] R. Luca. ZONES: A solution to the problem of dynamic screen formatting in CRT-based networks. Proc. 4th Data Communications Symposium 1.1-1.7, ACM/IEEE, October 1975.

140] H. Lycklama and D.L. Bayer. The MERT operating system. In Bell [21], pages 2049-2086.

141] B. Lyles. Ph.D. thesis, University of Rochester, in preparation April 1980.

Conference on Software Engineering 315-330, IEEE, September 1979.

114] S.T. Kent. Encryption-based protection protocols for interactive user-computer communications. Proc. 5th Data Communications Symposium 5.7-5.23, ACM/IEEE, September 1977.

115] S.R. Kimbleton and R.L. Mandell. A perspective on network operating systems. Proc. National Computer Conference 45:551-559, AFIPS, June 1976.

116] S.R. Kimbleton and P. Wang. Applications and protocols. To appear in Lampson [121].

117] S.R. Kimbleton, P. Wang, and E. Fong. XNDM: An experimental network data manager. Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks 3-17, Lawrence Berkeley Laboratory, August 1979.

118] S.R. Kimbleton, H.M. Wood, and M.L. Fitzgerald. Network operating systems -- An implementation approach. Proc. National Computer Conference 47:773-782, AFIPS, June 1978.

119] H. Kopetz, F. Lohnert, and W. Merker. An outline of project MARS: Maintainable Realtime System. Report 79-09, Computer Science Department, Technical University of Berlin, July 1979.

120] C.E. Krebs, C. Bumgardner, and T. Northwood. Terminal Transparent Display Language (TTDL)., Proc. National Computer Conference 45:365-371, AFIPS, June 1976.

121] B.W. Lampson, editor. Distributed Systems Architecture and Implementation: An Advanced Course. To be published by Springer-Verlag. Also Lecture Notes, Institut fuer Informatik, Technische Universitaet Muenchen, March 1980.

122] B.W. Lampson. Atomic transactions. To appear in Lampson [121].

123] B.W. Lampson. Case study: Ethernet, Pup, and Violet. To appear in Lampson [121].

124] B.W. Lampson and H.E. Sturgis. Reflections on an operating system design. CACM 19(5):251-265, May 1976.

125] K.A. Lantz. RIG User's Guide. Internal Memo, Computer Science Department, University of Rochester, March 1979.

126] K.A. Lantz. Annotated Bibliography on Distributed Systems. Internal Memo, Computer Science Department, University of Rochester, April 1980.

127] K.A. Lantz and R.F. Rashid. Virtual Terminal Control. In [185].

128] K.A. Lantz and R.F. Rashid. VTMS: A Virtual Terminal Management System for RIG. TR44, Computer Science Department, University of

98] R.C. Holt. Some deadlock properties of computer systems. Computing Surveys 4(3):179-196, September 1976.

99] R.C. Holt and M.S. Grushcow. A short discussion of interprocess communication in the SUE/360/370 operating system. SIGPLAN Notices 8(9):74-78, September 1973.

100] J.E. Hopcroft and J.D. Ullman. Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

101] A.L. Hopkins Jr. Fault-tolerant system design: Broad brush and fine print. Computer 13(3):39-45, March 1980.

102] J.J. Horning and B. Randell. Process structuring. Computing Surveys 5(1):5-30, March 1973.

103] J.G. Hunt. An introduction to LIMP: An experimental language for the implementation of messages and processes. Proc. 5th Conference on Programming Languages, Gesellschaft fuer Informatik, March 1978.

104] IFIP International Network Working Group 6.1. Proposal for standard virtual terminal protocol. INWG Protocol Note 91, February 1978. Also ISO TC97/SC16 N23, International Standards Organization.

105] C.H. Irby. Display techniques for interactive text manipulation. Proc. National Computer Conference 43:247-255, AFIPS, June 1974.

106] C.H. Irby. The Command Meta Language System. ARC Catalog Item 27266, SRI International, January 1976.

107] E.D. Jensen. The Honeywell experimental distributed processor: An overview of its objectives, philosophy, and architecture facilities. Computer 11(1):28-39, January 1978.

108] E.D. Jensen. Distributed control. To appear in Lampson [121].

109] E.D. Jensen. Operating system models. To appear in Lampson [121].

110] A.K. Jones. Protection mechanisms and the enforcement of security policies. In Bayer, Graham, and Seegmueller [20], pages 228-251.

111] A.K. Jones. The object model: A conceptual tool for structuring software. In Bayer, Graham, and Seegmueller [20], pages 7-16.

112] A.K. Jones, R.J. Chansler Jr., I. Durham, K. Schwans, and S.R. Vegdahl. StarOS, a multiprocessor operating system for the support of task forces. Proc. 7th Symposium on Operating Systems Principles 117-127, ACM, December 1979.

113] A.K. Jones and K. Schwans. TASK forces: Distributed software for solving problems of substantial size. Proc. 4th International

=> numports = number of ports owned by current process

SetBacklog (backlog, port)

Change the size of the primary queue for a port.

backlog = new queue size

port = port to change

Unlock (port)

Unlock a port so that messages can be received from it again. This routine is the complement of LockUp.

port = port to unlock

A.1.3 Emergency Messages

DisableEmergencies ()

Disable processing of emergency messages. This is done automatically by the Kernel whenever an emergency handler is about to be called, thus disallowing recursive handling of emergencies while in the handler.

EnableEmergencies ()

Enable emergency processing. This is useful, for instance, when a process wants to handle recursive emergencies; to do so, he must perform an EnableEmergencies within his emergency handler(s).

SendEmergencyMessage ([msg, dontfree]) --> sent

Send an emergency message -- i.e., a message to the EMERGENCYPORT of another process.

msg = message to be sent

dontfree = false --> release the data portion of the message after sending the message;
= true --> don't release the data

(default: false)

SetupEmergencyHandler (handler)

Set the emergency handler for the calling process.

handler = procedure to be used as handler

A.2 Second-level Primitives

A.2.1 Dynamic Port Allocation

AllocatePort () —> port

Allocate a "free" user port, i.e., one that is not currently being used for another connection.

=> port = port allocated

DeallocatePort (port)

Deallocate a user port.

port = port to deallocate

ReservePorts (p1 [, p2, ...])

Reserve ports such that they will not be dynamically allocated.

p1,... = ports to be reserved

A.2.2 Message Management

ReleaseMessage (msg)

Release any buffer arguments associated with message (strings and actual buffers).

msg = message whose argument to release

UnexpectedMessage (msg)

Generate an Error call of severity UTILITYERR in response to an "undesirable" message. If that message is an ERRMSG, then the error code will be the code from the ERRMSG; otherwise, the code is UNEXPECTEDMSG.

msg = the unexpected message

ValidateMessage (msg, id) --> valid

Validate a message as being of a particular type.

msg = the message to validate

id = message id

=> valid = true, message ok;
= false, otherwise

ValidateSender (msg, s1 [, s2, ...])

Assure the a message was from one of a set of processes. If not, generate an Error call with error ILLEGALSENDER.

s1,... = valid processes

=> if invalid sender, Error call: severity SYSERR, code ILLEGALSENDER

A.2.3 General Message-passing

Call ([msg, dontfree, allocateport, timeout]) --> response

Perform a remote procedure call — i.e., send a request to a process, wait for a response, and return that response.

msg = message template to use for communication
(default: @PGL)

dontfree = false --> release the data portion of the message after sending the message;
= true --> don't release the data
(default: false)

allocateport = false --> don't allocate a free port, use the one in msg
= true --> dynamically allocate a port (via AllocatePort) for

```
          the transaction
          (default:  false)

     timeout = # seconds to wait for completion of request
          (default:  forever)

  => response = data of the callee's response message, if  the  response
          is not an ERRMSG;
          otherwise, Call generates an Error call of severity CALLERR


Decline ([msg, error, dontwait]) --> sent

          "Decline" a request -- e.g., process A has sent a  message  to
          process B which B doesn't understand, so B rejects the request
          with an ERRMSG containing an appropriate error code.

     msg = request message template
          (default:  @PGL)

     error = error code for reply
          (default:  UNEXPECTEDMSG)

     dontwait = false --> suspend the sender if the receiver's  port  is
          full;
          = true --> don't suspend the sender
          (default:  false)

  => sent = true, message was sent
          = false, message was not sent


Reply ([msg, data, dontfree, dontwait]) --> sent

          Opposite of Decline -- i.e., having  processed  a  request  to
          completion, now reply.

     msg = request message template
          (default:  @PGL)

     data = reply data
          (default:  none)

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

     dontwait = false --> suspend the sender if the receiver's  port  is
          full;
          = true --> don't suspend the sender
          (default:  false)
```

```
  => sent = false, message not sent;
          = true, message sent
```

A.2.4 Connections

```
Close (connection [, dontfree, timeout]) --> closed

          Close a connection.

     connection = connection template

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

     timeout = # of seconds to wait for completion of the close
          (default:  forever)

  => closed = true, if connection closed
          = false, if, for instance, the request timed-out


Open (server [, openargs, dontfree, timeout]) --> connection

          Open a connection.

     server = service identifier (usually a string)

     openargs = argument buffer
          (default:  NULL --> no arguments required

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

     timeout = # of seconds to wait for completion of open
          (default:  forever)

  => connection = message template for connection, if opened
          = 0, if connection not opened


Read (connection, lvnumbytes [, buffer, timeout]) --> buffer,
          @lvnumbytes

          Read a block of data.
```

```
     lvnumbytes = address of the word containing the number of bytes  to
          read.   The address is used so that the actual number of bytes
          read will be stored back into the location.

     buffer = buffer into which to store the data.  If 0,  storage  will
          be allocated.
          (default:  0 --> allocate storage)

     timeout = # of seconds to wait for completion
          (default:  forever)

  => buffer = data buffer, if data read
          = 0, if no data read

     @lvnumbytes = actual number of bytes read


Write (connection, buffer [, numbytes, dontfree, timeout]) --> written

          Output a block of data.

     connection = process connection

     buffer = data block

     numbytes = number of bytes to output
          (default:  SizeMem(buffer)*2)

     dontfree = false --> release the data portion of the message  after
          sending the message;
          = true --> don't release the data
          (default:  false)

     timeout = # of seconds to wait for completion
          (default:  forever)

  => written = true, data written
          = false, not written (e.g., timed-out)
```

A.2.5 Process Synchronization

```
ReceiveSynch (sender [, timeout]) --> data

          Send a request for process synchronization and wait to receive
          it.

     sender = process sending arguments

     timeout = seconds to wait for data
          (default:  FOREVER)
```

=> data = data contained in process synchronization message

SendSynch (receiver [, data, dontfree, timeout]) --> sent

Wait for a request for synchronization and then send the data.

receiver = process to receive arguments

data = data to send
(default: no data, just synch)

dontfree = false --> release the data portion of the message after sending the message;
= true --> don't release the data
(default: false)

timeout = # seconds to wait for synch-request
(default: forever)

=> sent = data sent

A.2.6 Name Service

AssertName (name [, machine, procno])

Declare a process to the local name server.

name = name of process

machine = machine on which process resides
(default: machine on which calling process resides)

procno = process
(default: calling process)

Locate (procname [, lvmachine]) --> procno, @lvmachine

Locate a process given its name.

procname = name of process to be found

lvmachine = address of word into which to store machine of found process. If non-zero and contents non-zero, restrict the search to the specified machine.
(default: 0 --> don't save)

```
=> procno = process address, if found;
          = 0, if server not found

   @lvmachine = machine address of server, if found;
          = 0, if server not found
```

APPENDIX B

Keyboard and Control Functions

B.1 Keyboard

The Keyboard being designed will have a layout as depicted in Figure 19.

B.2 Control Functions

Four classes of control functions are of interest:

- screen and task management keys:
  - ABORTTASK
  - ALPHALOCK
  - AUTOBLOCK
  - BLOCKOUTPUT
  - CHANGECONFIGURATION
  - CHANGEIMAGE
  - CHANGEREGION
  - CHANGEVIEWPORT
  - DELETEIMAGE
  - DELETEMARK
  - DELETEREGION
  - DISCARDOUTPUT
  - INDIRECTINPUT
  - MARK
  - PASS
  - PICK
  - PUT
  - REFRESHSCREEN
  - RESUME
  - RESUMEMONITOR
  - RESUMESTASER
  - SUSPENDTASK
  - VIEW

- line-editing keys:
  - CURSORLEFT
  - CURSORRIGHT
  - CURSORTOENDOFLINE
  - CURSORTOSTARTOFLINE
  - CURSORWORDLEFT
  - CURSORWORDRIGHT
  - DELETECHARLEFT
  - DELETECHARRIGHT
  - DELETETOENDOFLINE

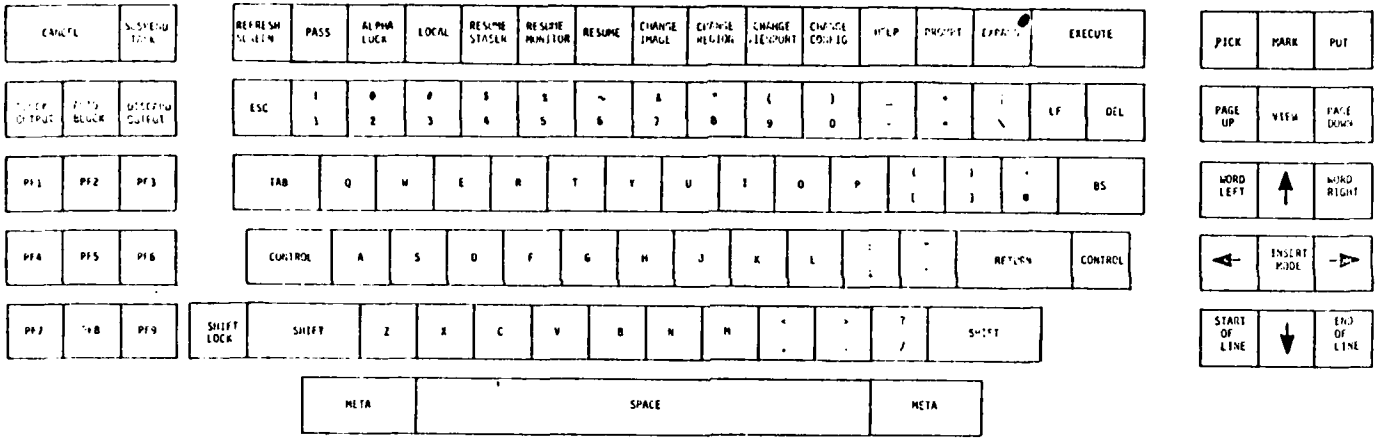


Figure 19. The RIGITS keyboard.

DELETETOSTARTOFLINE
DELETEWORDLEFT
DELETEWORDRIGHT
INSERTMODE

- page-editing keys:
  CURSORDOWN
  CURSORLEFT
  CURSORRIGHT
  CURSORTOENDOFLINE
  CURSORTOSTARTOFLINE
  CURSORUP
  CURSORWORDLEFT
  CURSORWORDRIGHT
  DELETECHARLEFT
  DELETECHARRIGHT
  DELETEPAGEDOWN
  DELETEPAGEUP
  DELETETOENDOFLINE
  DELETETOSTARTOFLINE
  DELETEWORDLEFT
  DELETEWORDRIGHT
  INSERTMODE
  JOINLINE
  PAGEDOWN
  PAGEUP
  SPLITLINE

- command-input keys:
  ASSIGNSLOT
  CANCEL
  COMMENT
  EXECUTE
  EXPAND
  HELP
  PROMPT

In the sequel, the syntax for control functions is <function> [<default Keyboard key>].

ABORTTASK [shift-CANCEL] - abort the task associated with the active VT

ALPHALOCK [ALPHA LOCK] - software shift-lock for the active VT (toggle)

ASSIGNSLOT [=] - assign a value to a command slot

AUTOBLOCK [AUTO BLOCK] - toggle the auto-block feature for the active VT

BLOCKOUTPUT [BLOCK OUTPUT] - block/unblock output to the active VT (toggle)

CANCEL [CANCEL] - "cancel" the current input

CHANGECONFIGURATION [CHANGE CONFIG] - change active Configuration

CHANGEIMAGE [CHANGE IMAGE] - change active Image

CHANGEREGION [CHANGE REGION] - change active Region

CHANGEVIEWPORT [CHANGE VIEWPORT]- change active Viewport

COMMENT [!] - in line-edit mode, treat remainder of the line as a comment

CURSORDOWN [down arrow] - move cursor down one line

CURSORLEFT [left arrow] - move cursor left one character

CURSORRIGHT [right arrow] - move cursor right one character

CURSORTOENDOFLINE [END OF LINE] - move cursor to end of current line

CURSORTOSTARTOFLINE [START OF LINE] - move cursor to start of current line (or field)

CURSORUP [up arrow] - move cursor up one line

CURSORWORDLEFT [WORD LEFT] - move cursor left one "word"

CURSORWORDRIGHT [WORD RIGHT] - move cursor right one "word"

DELETECHARLEFT [BS or shift-left arrow] - "backspace" one character

DELETECHARRIGHT [shift-right arrow] - delete the character at the cursor

DELETEIMAGE [shift-CHANGE IMAGE] - delete the active Image

DELETEMARK [shift-MARK] - delete a "mark"

DELETEPAGEDOWN [shift-PAGE DOWN] - delete a "page" of text downwards

DELETEPAGEUP [shift-PAGE UP] - delete a "page" of text upwards

DELETEREGION [shift-CHANGE REGION] - delete the active Region

DELETETOENDOFLINE [shift-END OF LINE] - delete to the end of the current line

DELETETOSTARTOFLINE [shift-START OF LINE] - delete to the start of the current line or field

DELETEWORDLEFT [DEL or shift-WORD LEFT] - delete one "word" to the left

DELETEWORDRIGHT [shift-WORD RIGHT] - delete one "word" to the right

DISCARDOUTPUT [DISCARD OUTPUT] - discard/don't-discard output to the active VT (toggle)

EXECUTE [EXECUTE] - "execute" a command; or confirm; or answer "yes"

EXPAND [EXPAND] - "expand" the current input

HELP [HELP] - display some tutorial help related to the current input

INDIRECTINPUT [@] - take input from a file

INSERTMODE [INSERT MODE] - toggle "insert" mode

JOINLINE [shift-up arrow] - DELETETOENDOFLINE, then append following line

LOCAL [LOCAL] - go into "local" mode in order to communicate directly with the Terminal

MARK [MARK] - place a "mark" in a pad (for pick and put)

PAGEDOWN [PAGE DOWN] - scroll a "page" downwards

PAGEUP [PAGE UP] - scroll a "page" upwards

PASS [PASS] - pass the next character as a "normal" character; i.e., don"t interpret it as a control function

PICK [PICK] - select the "marked" text

PROMPT [PROMPT] - display options, etc., related to current input

PUT [PUT] - place the last selected (picked) text into the Pad

REFRESHSCREEN [REFRESH SCREEN] - refresh the screen

RESUME [RESUME] - resume the last active Virtual Terminal

RESUMEMONITOR [RESUME MONITOR] - resume an instance of the Monitor

RESUMESTASER [RESUME STASER] - resume an instance of the Status Server

SPLITLINE [shift-down arrow] - split the current line, appending it to the following line

SUSPENDTASK [SUSPEND TASK] - suspend/resume the task associated with the active VT (toggle)

VIEW [VIEW] - detach/attach the viewing cursor from the output cursor (toggle)

APPENDIX C

Command Profiles

```
tool (command) id

login mode -- MUSTBELOGGEDON, in order to execute
          the command
     NEEDNOTBELOGGEDON

confirmation mode -- TRUE, confirm command
          FALSE

slot mode -- INTERPRETED, allow only those slots
                    specified in the CP
     UNINTERPRETED, allow literal slots (raw data)

help-text

# slots
slots:
     has-value flag -- FALSE --> no value;
           name sufficient
     needs-value flag -- e.g., switches may not
     confirmation flag
     required flag
     value-is-constant flag
     default
     prompts:
          verbose prompt
          terse prompt
     help-text

     internal slot name (code)
     external slot names (chain)
     type -- CHARACTER
          WORD, string bracketed by control or
               break characters
          STRING, string bracketed by quotes
          INTEGER
          BOOLEAN
          OLDFILENAME
          FILENAME
          NEWFILENAME
          PASSWORD
          LISTs of the above

     radix


     # sub-slots (switches)
```

dictionary for subslots
subslots : just like slots...

dictionary — external slot names, with references
to internal slot names

TIF — name of (disk) Process Profile for TIF

APPENDIX D

User Profiles

```
creation timestamp
modification timestamp
expiration date

protection status
user id (unique text string)
user number (unique integer)
RIG password

mailbox address

accounting info

access rights:
     tool list -- list of tool ids, the first entry
          being that tool to which the user is
          given access when he logs in
     site list — list of (site, password) pairs
     resource rights -- list of
          (resource, max, remaining) triples
     bound devices -- list of
          (device code, specific-device) pairs
          (e.g., (LPT, SUMEX:LPT))
     default directory
     file scopes and naming hierarchy (i.e., search
          order and composite directory)

command environment:
     case mode -- UPPER, ignore case
          UPLOW, treat cases as different

     herald (Exec prompt):
          mode -- VERBOSE
               TERSE
          verbose herald
          terse herald

     prompting:
          degree -- OFF, do not prompt
               PARTIAL, prompt only for
                    required slots
               FULL, prompt for all slots
          mode -- VERBOSE
               TERSE
          indentation -- > 0 —> go to next line
```

```
          and indent prior to prompt

feedback (expansion of keywords) mode -- VERBOSE
          TERSE

confirmation string (e.g., "[Confirm]")

control characters -- list of
     (function, char-string, echo-string) triples
     specifying defaults for control keys noted
     above

     (list of (device, list of (function...)) pairs
     may also be useful)

programmable function keys -- list of
     (character, program-string) pairs

for each accessible tool (indexed by tool id):
     autostop mode --
          ON, wait for SCROLL confirmation
               to scroll
          OFF
     command profile address --
          NULL --> use system default profile
          otherwise, disk address of Command Profile
                for this user
```

APPENDIX E

A Prototypical Emergency Handler

```
let Handler (msg) = valof
  [
     // "msg" is the request in progress at the time of
     // the call.  The emergency message is contained in
     // the process globals (accessed here by Emergency).
     // The handler should return a Boolean value
     // indicating whether or not to continue the request
     // in progress — false means abort the request.

     let event = Emergency>>Message.ID
     switchon event into
       [

       case TERMINATEMSG:
         [
            // Clean up state and terminate the process...
         ]
       endcase

       case PROCESSCRASH:
       case PROCESSSUICIDE:
         [
            let deadproc = Emergency>>Message.Data1
            test event eq PROCESSSUICIDE
             ifso
              [
                    // died naturally...do whatever
                 ...
              ]
             ifnot
              [
                   // died horribly ... do whatever
                 let howdied = Emergency>>Message.Data2
                 ...
              ]

               // Trying to communicate with dead process?
               // --> abort the send or receive in progress.

            if (deadproc eq msg>>Message.OtherProc) then
                 resultis false
         ]
       endcase

       ]

     resultis true
```

APPENDIX F

A Prototypical Process

```
let <1st-level-proc> (<initial-parms>) be
  [
     InitializePorts ()
     SetupEmergencyHandler (<emergency-handler>)

     let error = nil
     ErrorSet (SYSERR, lv err, <2nd-level-proc>,
               <initial-parms>)
     <quit-process> (error)
  ]


and <2nd-level-proc> (<initial-parms>) be
  [
     // Initialize process with <initial-parms>.

     .
     .
     .

     let msg = vec MESSAGESIZE
     let msgcopy = vec MESSAGESIZE
     let error = nil

       [
          ReceiveAny (msg)

       // The copy is made so that lower-level routines may
       // munge the message template without losing track
       // of the initial arguments.

          BlockMove (msg, msgcopy, MESSAGESIZE)
          let ok = ErrorSet (PROCERR, lv error,
                             <3rd-level-proc>,
                             msgcopy)

       // Release the message arguments --
       // e.g., strings and buffers.

          ReleaseMessage (msg)

       // If we had an error and the message id indicates
       // the process expects a response, "decline" the
       // request.
```

```
          unless ok % <is-known-not-to-want-a-reply> then
               Decline (msg, error)
        ] repeat
  ]


and <3rd-level-proc> (msg) be
  [
     .
     .
     .

     switchon msg>>Message.ID into
       [

       case <id1>:
         [
         // ValidateSender will perform an Error call if
         // the sender of the message is not one of those
         // specified.

            ValidateSender (msg, <sender1>, <sender2>,...)

         // Process the request.
            .
            .
            .

         // Send an appropriate reply if necessary.

            Reply (msg, ...)
         ]
       endcase

     .
     .
     .


       default:
         [
         // Avoid infinite ERRMSG loops -- i.e., I send
         // an ERRMSG to process B, who doesn't know what
         // it means, so he sends an error message to me!

            unless (msgid eq ERRMSG) do
                 Error (PROCERR, UNEXPECTEDMSG)
         ]
       ]
     .
     .
     .
  ]
```

```
and <emergency-handler> (msg) = valof
  [
     // See Appendix E.

     switchon Emergency>>Message.ID into
       [

       case TERMINATEMSG:
         [
            <quit-process> (NOERR)
         ]
       endcase


       case PROCESSSUICIDE:
       case PROCESSCRASH:
         [
         // process with which you have registered has died
         // --> do something
             .
             .
             .
         ]
       endcase

       ]

     resultis <whether-want-to-abort-request-in-progress>
  ]


and <quit-process> (error) be
  [
  // Clean up state -- e.g., close open files and
  // Virtual Terminals, or kill your descendant processes.
     .
     .
     .

  // Kill yourself.

     KillProc (MyID(), error)
  ]
```

END

FILMED

7-85

DTIC